MIT Concurrent VLSI Architecture Memo 40

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

MDP Programmer's Manual[1]

Michael Noakes[2]

Abstract

The Message Driven Processor, MDP, is a VLSI implementation of an integrated processor core for fine-grained parallel computers. The processor is to be used to construct an experimental massively parallel computer; the MIT J-Machine. Primitive mechanisms are embedded to support a wide variety of programming models including a message passing organisation, shared memory applications, data-parallel programming, and dataflow applications. Target applications include CAD design and simulation, physical modeling, transaction processing, graphics rendering, parallel language design, architectural evaluation.

This document attempts to address the needs of a wide range of potential readers. The early chapters focus on the requirements of new users with little exposure to the principles and motivations of the MDP or the J-Machine parallel computer. The later sections focus on details of the chip; the format of the registers, special features and so. The document concludes with the use of the current assembler MDPSim and examples of code fragments to clarify the discussions. It is also intended to serve as a complete reference manual for the MDP and as such supersedes the previous such document, "Message-Driven Processor Architecture Version 11".

Keywords: Processor Architecture, VLSI, Parallel Processing, Message Driven Processor, Fine-Grain, Networks, Cache, Concurrent Smalltalk.

[1]This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The research described in this paper was supported in part by the Defense Advance Research Projects Agency of the Department of Defense under contracts N00014-80-C-0622 and N00014-85-K 0124 and in part by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation and IBM Corporation.

[2]Includes portions of CVA memo 14

Contents

Chapter 1

ARCHITECTURAL OVERVIEW

The Message Driven Processor, MDP, is a VLSI implementation of an integrated processor core for fine-grained parallel computers. The processor is to be used to construct an experimental massively parallel computer; the MIT J-Machine. Primitive mechanisms are embedded to support a wide variety of programming models including a message passing organisation, shared memory applications, data-parallel programming, and dataflow applications. Target applications include CAD design and simulation, physical modeling, transaction processing, graphics rendering, parallel language design, architectural evaluation.

This document attempts to address the needs of a wide range of potential readers. The early chapters focus on the requirements of new users with little exposure to the principles and motivations of the MDP or the J-Machine parallel computer. The later sections focus on details of the chip; the format of the registers, special features and so. The document concludes with the use of the current assembler MDPSim and examples of code fragments to clarify the discussions. It is also intended to serve as a complete reference manual for the MDP and as such supersedes the previous such document, "Message-Driven Processor Architecture Version 11".

1.1 OVERVIEW

The Message Driven Processor includes on a single chip:

- 32 bit integer core
- On-chip 4 Kword static ram
- Three dimensional router and integrated network interface
- Multiple register banks for fast context switching among three priorities of execution
- Tag support for runtime type checking and for synchronisation primitives
- Fast trapping mechanism for exceptional situations
- Segment-based address unit
- Name cache for the support of a global name space
- Support for up to 1 Mword of external DRAM including Error Correction coding, dynamic refresh, and both static-column and page-mode memory devices.
- Low-level diagnostic interface for system initialisation

To minimise contention for the shared on-chip static ram, the major modules read and write 4 words of this memory in a single cycle. The core arithmetic and logical instruction operate in a single clock cycle. Multiple cycle operations are controlled by a hardware state machine controller to free the programmer from the need to monitor pipeline dependencies in software. The instruction set of the processor follows conventional practices for streamlined execution while balancing the need for increased code density on fine-grained processing nodes.

As the name suggests, the major activities of the MDP are scheduled by the reception of messages from within the J-Machine. Messages are routed through to a neighboring node or queued for the current node without intervention of the integer core. When the integer processor completes the operations associated with the current thread, the next message is dispatched efficiently under hardware control. Messages can

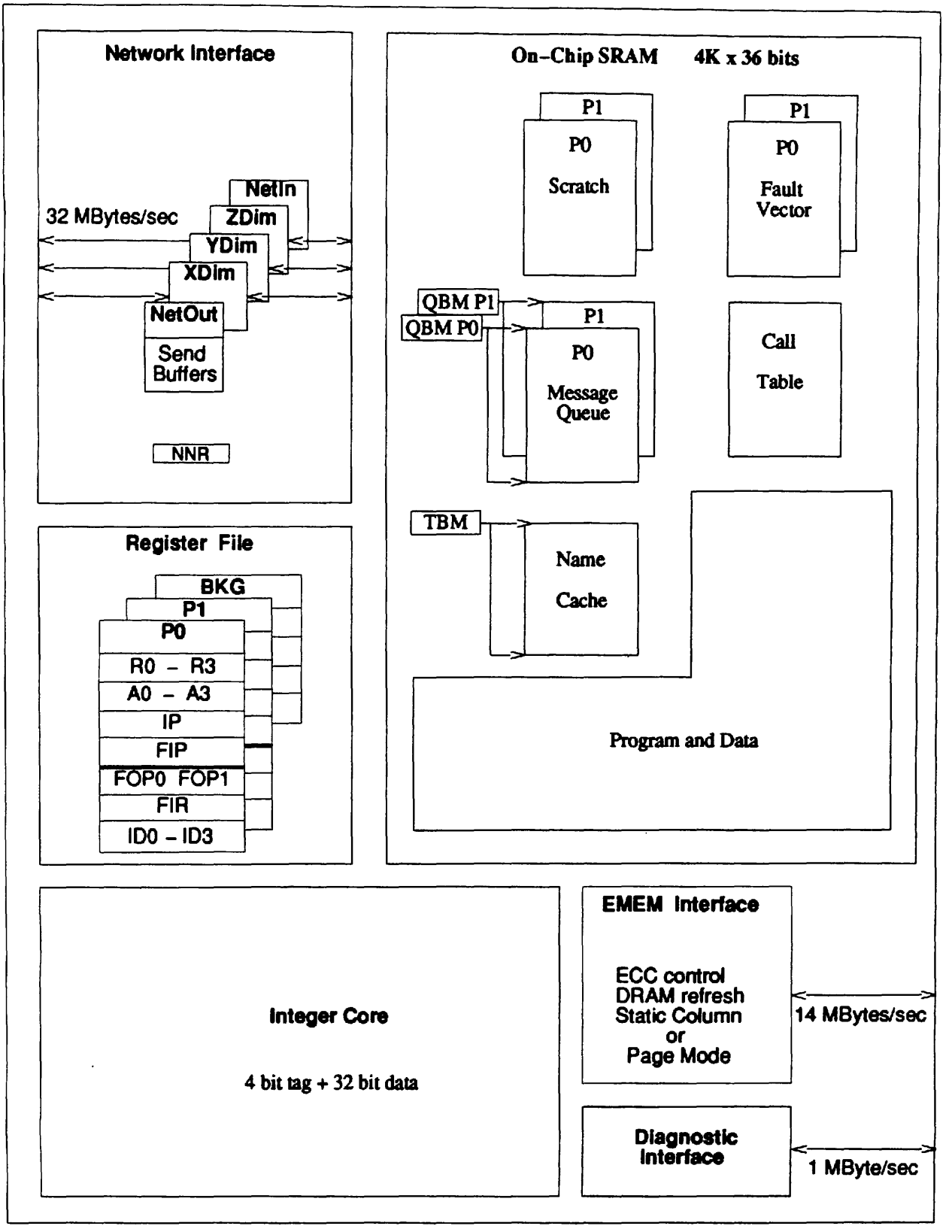


Figure 1.1: Major Chip Modules.

be processed at one of two priorities and independent register banks provide rapid transitions between the priorities. In addition, there is a background priority that executes whenever the two message queues are empty. The major functional blocks are indicated in figure 1. Typical data rates are indicated for the main external interfaces assuming a 16MHz processor clock. These values are provided for expository purposes and do not necessarily correspond to the final operational clock rate of the J-Machine.

1.2 INTEGER CORE UNIT

The integer core includes both the arithmetic and logical execution unit and the addressing unit that supports a base-and-bounds segment addressing protection scheme.

The instruction set includes

- Arithmetic and logical operations on operands in the general purpose register file; additionally one of the source operations can be from memory. Load and store operations between memory and the data registers.

- Transfers between the data registers and the address registers, the network control registers, the name cache control registers, and the status registers. It is also possible to transfer operands between registers in different priorities.

- Flow of control instructions for both short and long offsets, and an efficient call mechanism for low-level primitive routines.

The integer processor operates on tagged data; every word of the machine is tagged with a 4 bit value that indicates the type of the data stored there. Although these type tags are always monitored by the processor core, a section of code may inhibit the triggering of tag exceptions at the discretion of the programmer. In this way, the programmer may choose to perform a sequence of "non-standard" operations on the tagged values without regard to the standard type error processing. This mode of execution will normally be selected for reasons of efficiency within the low-level core of a systems oriented programming application. Tags are also used to provide fast trapping for two primitive synchronisation types supported by the hardware.

In addition to the type checking performed by the hardware, structure references are normally bounds-checked by the processor. A physical address consists of a base and a length and indexed accesses into the vector defined by this tuple are bounds-checked in parallel with the initiation of the access.

1.3 NETWORK INTERFACE

The network interface supports the injection, transmission, and reception of messages within the machine. Messages are injected using a set of SEND instructions. These instructions implement all of the standard addressing modes of the instruction set thereby simplifying the injection of message arguments. Messages are transmitted across the network independently of the processor core using the network routers integrated into every MDP. A message may be injected at one of two priorities. Messages at the higher priority receive preferential routing and make use of independent buffers. When a message reaches its destination, it is added to the tail of a queue associated with the priority of the message. Messages are scheduled for execution efficiently by the hardware whenever the associated priority is ready and certain special control flags permit.

The network control registers and queue memory is directly accessible by user code. The programmer may control the location and length of both message queues. Each queue may be up to 1024 words and must be located in on-chip memory. The general trap mechanism is used to support flow-control, by inhibiting the injection of messages when the network in congested, and to trap to system code to handle queue overflows.

1.4 NAME CACHE

A portion of the on-chip memory may be configured as a general purpose translation table. Data is entered explicitly with an associated key and, in the absence of later hash collisions, may be extracted efficiently using that key. One of the more obvious applications of this mechanism is the support of a global namespace for objects in the machine. This table is implemented as a two-way set associative cache with explicit enter and lookup operations. It can be configured to contain up to 512 key-value pairs.

1.5 DIAGNOSTIC INTERFACE

The diagnostic interface is an external interface between the processor core and the host machine. It will be discussed in this manual as it is critical to the initiation and execution of core programming systems. It supports the ability to download the initial program core, to inspect the contents of memory, to start and to stop the execution of the processor.

1.6 SOFTWARE DEVELOPMENT

There are currently two primary programming environments for the J-Machine; assembly language and Concurrent Smalltalk (CST). Other systems are under development including Concurrent Aggregates and Dataflow computation. There are two simulators for small assemblies of processors; the instruction-level simulator MDPSim, and the register-transfer level simulator mdp_test. An interpretive 'scripting' language and monitor, Jmon, provides uniform interactive use of a J-Machine or of the simulators.

1.7 THE PROTOTYPE J-MACHINE

The initial prototype of the J-Machine parallel computer will consist of at least 1024 MDP's connected by a high-performance 3 dimensional mesh network. Each processing node can include up to 1 Mword, 36 bit words, of private memory; for the initial J-Machine this is currently limited to 260 Kwords. The machine will include additional nodes that support peripherals such as graphics displays and disk drives.

Chapter 2

PROGRAMMING MODELS OF THE MDP

One of the difficulties of describing the organisation and operation of the Message Driven Processor is that it has been designed to be a platform for implementing a range of parallel programming models in the context of a general purpose fine-grained parallel computer. The design seeks to identify and implement core mechanisms that are expected to be valuable to the efficient operation of many of these possible programming models. The design decisions have evolved in a manner that tends to shape the lowest levels of these systems into a particular form of interaction, but care has been taken to ensure that sufficient flexibility has been retained to allow explorations in the neighborhood of this style.

A consequence of this flexibility is that the processor includes several modes of operation, and that the personality of the program becomes subtly different as the user selects among the modes. Typical programs will use each of the modes at different times during the execution and therefore most users need to understand the use of the major control registers and flags.

The different characters of the processor complicates an introductory discussion of the features provided because there seem to be so many exceptions to be described and appreciated. Even the apparently simple task of defining the operation of the ADD instruction is hampered by the different behaviors elicited by the current mode of operation. Rather than attempting to simply enumerate the list of registers, instructions, and modes while noting the different behaviors obtained in each mode, we will discuss the major styles of operation and the manner in which the basic mechanisms support these programs. The initial discussion will focus on the standard low-level programming model that the processor was designed to support. The alternative behaviors will be described in detail later. Chapter 3 summarises, integrates, and unifies the description and should be used as the programmer's reference once the basic operation is understood.

2.1 THE MESSAGE PASSING MODEL

The Message Driven Processor includes a number of mechanisms designed to assist in the development and debugging of higher-level programming systems on a fine-grained parallel computer. Fundamental among these is the tight coupling between the processor core and a high performance communication network, efficient hardware dispatching of threads, use of tagged words to provide dynamic type-checking and to support synchronisation primitives, the use of segment-descriptors to provide bounds checked access to memory 'objects', the provision of fast-trapping to user/system code to support extended data types and exceptional cases, and rapid context switching.

It is important to note that the MDP is designed as a fine-grained node in a parallel computer. This means that it should be possible to build a complete processing node from just a few chips, thereby allowing many nodes to be placed in a small volume, and that the processor be able to execute sequential threads of just 50 - 100 instructions. A typical application is expected to partition the data set into many thousands of relatively small objects that can be distributed across the nodes of the machine thereby enhancing opportunities for parallel access.

2.2 THE NODE ID

Every processor in the J-Machine has a unique node number that is assigned during system initialization. The node id is a 16 bit integer that indicates the <x y s> cartesian coordinates of the node within the cube. The representation supports up to 32 nodes in each of the x and y dimensions, and up to 64 planes in the

s dimension. The node number is stored in the Node Number Register, NNR, of the MDP. It must also be 'programmed' into the routers during system initialisation as described later.

2.3 OBJECTS

It is important to appreciate that the use of the words 'object' and 'message' to describe the operation of the MDP do not refer to precisely the same concepts as these same words when applied to certain higher-level languages such as C++, Smalltalk, or CLOS. There is certainly a common thread to the ideas, and the mechanisms supported by the MDP have been influenced by the style of interaction advanced by this increasingly popular model of program organisation. However, the use of the terms on the MDP necessarily refers to more primitive mechanisms.

The term object, or segment, or vector, or structure, is simply a bounded sequence of tagged words. Access to an object is achieved through an address, or segment descriptor, located in one of several special purpose registers on the MDP. An address indicates both the base and length of the object. All accesses are bounds checked and a trap to a user specified routine is invoked if the bounds are exceeded.

A runtime system may support the relocation of certain classes of objects. This might consist of copying the object to another location within the current processor or may involve the migration of the object from one MDP to another. In such a system, it will not be appropriate to pass physical addresses of the objects among the nodes or even to store an address within a single processor for an extended time. Instead it is preferable to generate a name, or handle, for the object and to use this in all exchanges. System tables are then used to translate the name to a physical address as needed. One common scheme is to dictate that every object is owned by a particular node; the home node. The node number of the owner is combined with a unique index among all the objects owned by that node as the name of the object. The home node is then responsible for knowing the current location of its objects and forwarding requests if the object has been relocated to another node.

There are many possible enhancements on the use of objects. It is desirable to copy certain immutable objects, e.g. code blocks, to other nodes thereby increasing the parallel bandwidth to the object and to enhance locality. It is also possible to create a distributed object. This type of object is no longer a simple vector of words but is instead partitioned into a number of constituent parts which may be distributed around the machine. Once again, the goal is to increase the opportunity for parallel access to the object.

2.4 MESSAGES

All communication and coordination among the processing nodes occurs as a result of the exchange of messages among them. Sending a message from one node to another has two effects; it communicates data between the nodes, and it schedules work to be done on the target node. A message is a small data object consisting of a header word that indicates the initial entry point for the code to be executed and a number of arguments that will be needed by the computation. The length field of the header word of the message also indicates the number of arguments supplied. This field is used to validate all accesses to the arguments of the message. The request might be something simple such as to lookup the value of a slot in a data object stored in the memory of the target node, or it may initiate an extensive sequence of calls and secondary message sends.

Consider the simpler case just mentioned; a message sent to request the value of a slot in a data object. This message might be of the following form

MSG	GET-SLOT 5
ID	NAME OF OBJECT
INT	OFFSET
ID	OBJECT FOR RESULT
INT	OFFSET

The first word of the message contains the physical address of the start of the code within the memory of the target processor, and the length of this message. The second word, the first argument, is a reference to the object. This could be the physical address of the start of the object in some simple programming systems, but in general we will think of this as a unique name for the object known to both the source of the message and the target. This name will be translated into a physical address by the destination. The second argument is simply the offset desired within the object. The third and fourth arguments indicate where the result should be sent.

Messages sent between nodes travel in a deterministic path between the nodes until the destination is reached. This path is so-called "Manhattan Routing". Messages travel in the x-dimension, then the y-dimension, and finally the z-dimension. There are two possible priorities for a message; priority zero and priority one. The priority is assigned by the sending node and priority one messages receive preference during routing. When the message reaches the destination it is stored in the appropriate message queue by the network interface without intervention by the integer core. The message queues are implemented as circular buffers. The size and location of each queue is determined by the initialisation routines and can be altered at runtime if necessary although this is not expected to be common.

2.5 PRIORITIES

The processor can execute instructions in one of three possible priorities; the background, priority zero, and priority one. Many of the machine registers are replicated for each priority thus permitting the processor to switch rapidly among these priorities without saving state.

The processor is initially in the background. If a message arrives at priority zero or priority one, the processor will switch from the background to the appropriate priority. The instruction pointer will be set from the header word of the message and one of the address registers, explained in a moment, will be set to point to the message. The processor will then begin executing instructions. If the processor is executing at priority zero and another priority zero message arrives, it will be added to the priority zero message queue. When the processor completes the current message, by executing the SUSPEND instruction, the message will be removed from the queue and the next message will be processed. If the processor is executing at priority zero and a priority one message is received, the processor will switch to priority one and run the new message. When that messages completes, the MDP will resume the priority zero message.

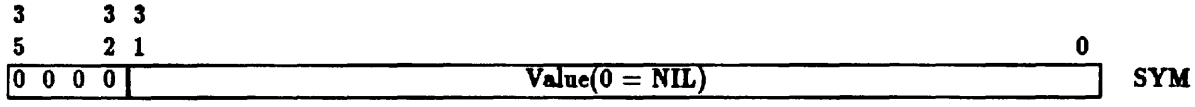
The current priority is indicated by the state of 2 bits; the B-bit and the P-bit. The B-bit is set when the MDP is executing in the background. If the B-bit is clear and the P-bit is clear then the processor is in priority zero. If the P-bit is also set then the processor is in priority one.

2.6 TYPES

The Message Driven Processor incorporates explicitly tagged data values in both memory and the processor core. Words in memory and in most of the machine registers consist of 36 bits; 32 bits represent the value, and 4 bits indicate the type. There are only 13 distinct types; four of the possible types are proper subsets of the instruction type.

2.6.1 Symbol

A symbol contains an atomic value. Symbols can only be compared with each other to determine whether they are the same symbol or not. There is a special symbol NIL in which the data portion is all zeros.

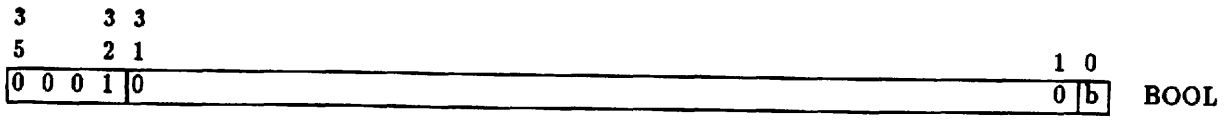


2.6.2 Integer

This type is the conventional 2's complement integer in the range -2^{31} to $2^{31} - 1$. A full set of arithmetic, logical, and comparison operations are defined on integers.

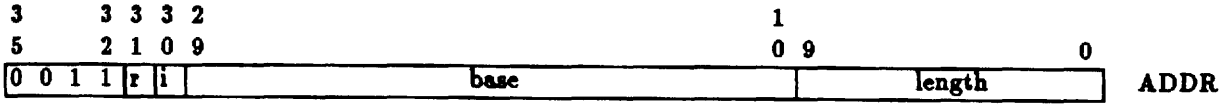
2.6.3 Boolean

There are two members of this set; TRUE and FALSE. They are distinguished by the least significant bit in the word. All the other locations should be zero. This type most commonly arises as the result of comparison operations. They may also be used as operands to logical operations.



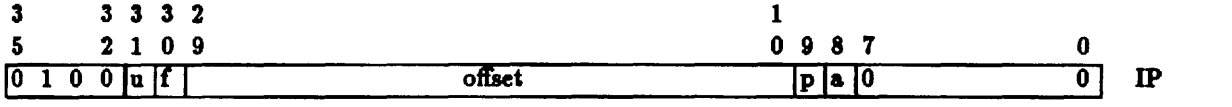
2.6.4 Address

The address type contains a base/length pair and two single bit fields. Indexed accesses relative to an address value are bounds checked against the length field of the word. If the length field is zero, then bounds checking is not performed. The 20 bit base field supports a 1 Mword address space, and the 10 bit length field permits bounds checked segments of between 1 and 1023 words. A length field of zero indicates an object of unspecified length. The r and i bits are explained later.



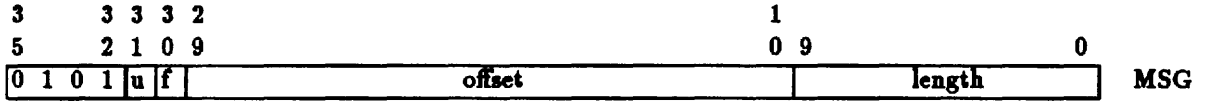
2.6.5 Instruction Pointer

This data type is appropriate for loading into the instruction pointer register either directly or as the result of an exceptional situation or call. The offset field is analogous to the base field of an address, i.e. provides for accesses into the 1Mword address space of the MDP. The other fields are explained in more detail later.



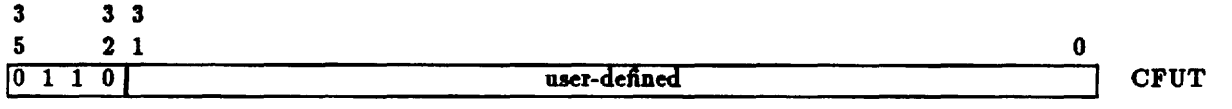
2.6.6 Message

This type is stored in the first word of a message. The upper fields of the header are loaded directly into the instruction pointer as part of the dispatch mechanism. The length field indicates the length of the message including the header word itself. The u and f bits are explained later.



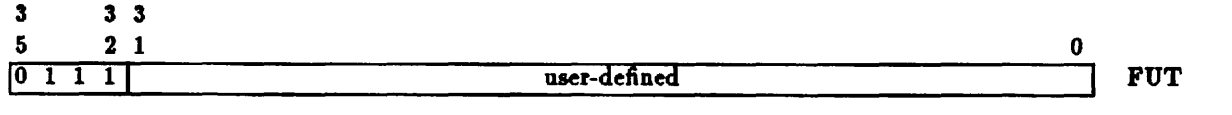
2.6.7 Context Future

This is a special marker used to support efficient synchronisation of parallel tasks. It is stored into structures prior to forking a parallel thread and, in normal use, forces an exception if the location is read again before the expected value has been determined. The trapping behavior is designed to prevent cfut-tagged values from migrating to another node or being stored in general data-structures.



2.6.8 Future

This is a more general synchronisation marker. It is expected to be used to indicate a first class data structure; one that can be stored in structures, passed as an argument to a function, and so on.



2.6.9 User Defined

The types TAG8, TAG9, TAGA, TAGB indicate user defined data types of unspecified format. Primitive operations on these types provide a direct trap to a unique handler for each type.

35	32	31	0	
1000		user-defined		TAG8
1001		user-defined		TAG9
1010		user-defined		TAGA
1011		user-defined		TAGB

2.6.10 Instruction Type

The types INST0, INST1, INST2, INST3 represent instruction types. This representation permits two 17-bit instructions to be stored in each word by overlapping the low two bits of the type with the high two bits of the first instruction.

35	34	33	17	16	0	
11	00	first instruction		second instruction		INST0
11	01	first instruction		second instruction		INST1
11	10	first instruction		second instruction		INST2
11	11	first instruction		second instruction		INST3

2.7 THE DATA AND ADDRESS REGISTERS

The MDP contains four general data registers, R0-R3, and four address registers, A0-A3, for each of the three priorities. The four general data registers are 36 bits long and can store any object of any type. R0 is used for the creation of long constants and is therefore a little less useful for storing computation temporaries than the other three registers.

The four address registers only store addresses. Although they are effectively 36 bits long, the upper four bits always read as ADDR. It is generally considered to be an error to attempt to write a non-address type into an address register. An attempt to do so will either cause a trap or simply be ignored depending on the mode of the processor.

When a message is dispatched, address register A3 is set to the start of the message. This allows the code to read the arguments contained in the message. It is possible to use A3 for other purposes if the computation no longer requires the contents of the message. However this requires certain flags to be adjusted appropriately as will be discussed later.

Address register A0 is used as the base register for instruction fetching and may be used for reading and writing data. The special uses of this register will be described in more detail later.

2.8 HANDLING A SIMPLE MESSAGE

Consider again the case in which a message is sent from one node to another to request the contents of a particular slot in an object. The message will be stored in the appropriate message queue, P0 or P1, on the target processor until the MDP is ready to execute it. At that point, address register A3 is initialised so that it points to the next message in the message queue, the length field is extracted from the header of the message and loaded into the length field of A3, and the instruction pointer is loaded from the physical address stored in the header. The code to be executed might be of the following form:

```
get-slot:   ;;; Extract the value from the object
            move        [1, a3], r2     ; Fetch the name of the segment into r2
            move        [2, a3], r3     ; Fetch the offset into r3
            xlate       r2, 0, a1       ; Translate the name to the physical address
            move        [r3, a1], r3    ; Fetch the data at the required offset

            ;;; Generate the header word of the message, an immediate constant
            dc          MSG:(reply_value << addr_base_pos) | 4
```

```
;;; Extract the name of the object waiting for the result
move        [3, a3], r1      ; Fetch the name of the receiver

;;; We can use this value directly but we demonstrate bit-masking here
wtag        r1, INT, r2      ; Convert it to an integer
and         r2, $FFFF, r2    ; Mask off the low 16 bit NNR

;;; Send the reply at priority zero
send2       r2, r0,  0
send        r1, 0
send        [4, a3], 0
sende       r3, 0

;;; Done
suspend
```

This code has been written using the syntax of the assembler that is incorporated into the MDP instruction level simulator MDPSim. Later sections of this document describe the instruction set of the MDP more fully and the use of MDPSim.

The first two move operations extract the name of the object and the offset within it and store them in the general registers R2 and R3. The xlate operation performs an associative lookup within the local name cache to translate the name to the physical address of the object. If the name is not found in the cache, either because the name is bad or because the cache entry has been reused, the MDP will trap to the xlate-fault handler. This handler will then attempt to resolve the name using a set of system tables. The handler may choose to use the instruction's second argument, a constant integer between 0 and 3, to assist in the search for the key. In either case, if the name can be translated, the result will be placed in A1. Finally the value is fetched and stored in R3. Note that this instruction will fault if the specified offset is too large for the referenced object. This error might cause the processor to send an error warning to the console and then idle to permit the programmer to inspect the state of the MDP at the time of the error.

Generating a reply message consists of SENDing a sequence of words to the network interface. A strength of the MDP is that the send instructions are normal operations in the processor that can utilise all the standard addressing modes of the instruction set. The first word to be sent is the NNR of the destination. The remaining words are the header and then arguments of the message. The instruction set allows short integers and certain special constants to be generated and moved into any general register in one instruction but long constants cannot be formed as easily. Instead the DC instruction is used to define an inline long constant and the resulting value is stored in R0. In this case we have formed a long constant whose tag field is the type MSG, and that contains the physical address of the reply code and the length of this message in the appropriate fields.

We assume that an object ID consists of the home node number in the low 16 bits of the name, and an integer unique to that object on that node in the high 16 bits as discussed earlier. This format is designed to permit it to be used directly as the destination since the send instruction ignores all but the low 16 bits in the first word for a message and only traps when the data is of type CFUT. However, the code shown demonstrates the use of bit-masking for expository purposes. First the name is converted to be of type INT using the wtag instruction, to avoid a type-fault when the bit-masking is done. We are careful to leave the name in R1 and compute the node number in R2. We note that the wtag instruction and the bitwise-and instruction have a similar format. The first operand is a general data register as is the destination. The second argument is more flexible. For the wtag instruction we have exploited the ability to refer to any small constant in the range -16 to 15 directly. The AND instruction has exploited the ability to refer to one of the 8 special immediate constants.

The destination node number and the message header, in R2 and R0 respectively, are sent to the network interface at priority 0 using a single instruction. The next instruction sends the name of the object awaiting the result value and the offset where the result should be written. The final send instruction transfers the value itself and indicates that there are no more arguments. The processor then executes the suspend instruction which pops the message from the queue and permits the MDP to initiate the next available message.

Note again that the header of the message must contain a physical address on the target processor. In general this is achieved by ensuring that all the message handlers are placed in exactly the same place on every node. Beware of a common error in which two programs that are almost the same and have the same set of labels, but at slightly different offsets, are placed on two different nodes. It is also possible to develop a system in which a few core routines are common to all the nodes and in which nodes that need to establish

access to specialized routines can pass the needed addresses dynamically during initialization. Ultimately, changes to the linker can be used to support repeated but specialized needs. In practice, this issue is rarely the problem that one might imagine so long as one is aware of the problem.

In this very simple example, we have encountered many of the basic features of the MDP. We have seen the handling of a simple message, have seen that typical instructions can use arguments stored in a general register, in an object pointed to by an address register, or a special constant. We have seen possible situations in which a trap can occur.

2.9 A0-RELATIVE ADDRESSING

We have noted that the MDP supports the use of relocatable objects and have even specified that code blocks can also be objects, and yet the example message handler just discussed clearly did not take advantage of this feature. In fact we went to great lengths to explain that one had to exercise appropriate care to ensure that the correct physical address was stored in the message header. We will now see how to exploit the use of relocatable code blocks and A0-relative addressing to enhance the use of message handlers.

It is expected that many 'interesting' applications will contain significantly more code than can be stored at one time on a single node. It is then necessary that the programming system be able purge local memory of user code that is no longer likely to be needed and copies of the required blocks be brought to the node. In such a system, the majority of the code blocks are implemented as objects. The lowest level of the runtime system, which is identical on all nodes, translates the name of a function to a pointer to the current location of the code object and then jumps to the start of the code. Consider a user-level function to compute the dot-product of two vectors. This function is to be invoked on a node by sending a message of the form

MSG	APPLY-FUNCTION 6
ID	DOT-PRODUCT
ID	VECTOR 1
ID	VECTOR 2
ID	CONTEXT FOR RESULT
INT	OFFSET

When this message is received, the kernel code for apply-function is called. This translates the name DOT-PRODUCT into a pointer to the code block. If the code block is not currently available on this node, the runtime system will request that a copy be sent to the node and then suspend the task. When the copy arrives, the function application will proceed.

We have already noted that A0 is used as the base of all instruction fetches and can also be used in data reads and writes but this did not appear to play a role in the first example. It will explicitly do so in this case. Register A0 can be used in two ways; a0-absolute and a0-relative. In the former mode, any reference through A0, whether implicitly as for instruction fetching or explicitly when used to fetch an instruction's operand, ignores the actual value in A0 and treats it as though it points to a segment at the start of memory and of indefinite length. In this mode the offset stored in the instruction pointer is the absolute value of the next instruction word to be fetched. If the user selects A0-relative mode, then A0 points to the beginning of a code block and the offset in the IP is a small integer relative to the beginning of the code block. When a message is dispatched, the processor is set to A0-absolute mode and the offset stored in the header of the message is stored in the IP. Let us examine a representative version of the code to implement apply-function.

```
apply-function:
        ;;; Get the pointer to the code block
        move        [1, a3], r0
        xlate       r0, 0, a0

        ;;; Jump to the start of the code within the code object
        dc          ip:(Foffset << ip_offset_pos)
        move        r0, ip
```

This code is very close to the code used by the Concurrent Smalltalk runtime system COSMOS. The first two instructions set A0 to point to the code object. If this object is not present, the xlate-fault handler will request a copy of the code and then suspend this task. Altering A0 does not affect the execution of the current code because its value is being ignored for both instruction and data accesses. The third instruction prepares a pointer of type IP with an offset to the start of the code within the code object i.e. a small integer offset based on the data format of a code object. The a0-absolute flag is clear in this new word and so, when it is finally loaded into the IP, the processor begins executing code at the appropriate offset within the code object. Now any data reads or writes of the form

```
add         r0, [r1, a0], r2
```

will be relative to the code block rather than to the absolute address contained in r1.

2.10 A3-RELATIVE ADDRESSING

We have seen that when the MDP dispatches a new message, A3 is set to the start of the message. We can then use this pointer to access the arguments. There are two slightly subtle issues to consider. We know that messages are stored in a circular buffer. What happens when the new message does not quite fit in the region left between the current queue pointer and the end of the buffer? Also, what happens when a message that has been suspended and saved in main memory is finally resumed? These two questions are somewhat related.

Let us consider the first question; the case in which the message "wraps-around" the message queue. Words of the message are collected and stored normally until the end of the buffer is reached and then the remaining words are stored at the start of the buffer. However it is clear that the user does not want to deal with this issue and so the MDP provides the illusion of a sequential set of words. Accesses through A3 are compared against the base and length indicated by the register that points to the entire message buffer, and accesses past the end of the queue are translated to the appropriate offset at the start. This is achieved at the same speed as a standard memory access.

Now we examine the second issue; that of resuming a suspended thread. Consider the previous example in which a message to invoke dot-product was processed. If the code block is not available on the current node, then the message must be copied from the message queue into another area of the MDP memory so that the arguments will be available when the code is delivered. The MDP issues a request for the code object and then suspends the current message which causes it to be removed from the message queue. Later, perhaps after the MDP has executed dozens of other messages, a reply arrives delivering the code object. The code object itself is copied from the message and then the original thread is resumed. A3 currently points to the message that provided the code but this code expects it to point to the original one. We must therefore adjust A3 so that it points to the current location of the message. However, accesses via A3 will now point beyond the end of the queue and so we must prevent the wrap-around feature from being used.

The Q-flag provides exactly this control. When it is set, as it is during a message dispatch, accesses relative to A3 are checked against the bounds of the message queue. When it is cleared, the bounds of the queue are ignored. It is clear that care must be taken during the interval in which A3 is to be redirected and Q is cleared.

2.11 FAULTS

The MDP defines 19 fault conditions and orders them such that a single handler will be selected if more than one fault occurs at the same time. When a fault is detected, such as a type-fault or an xlate-fault, the current state of the computation must be saved and then the appropriate handler must be selected.

There are four registers associated with a fault. The Faulted Instruction Pointer, FIP, stores the value in the IP at the time of the fault. This will be used to resume the instruction sequence when the fault has been handled. If the fault is associated with the execution of an instruction, then the current instruction is stored in the Faulted Instruction Register, the FIR, and the Faulted Operand registers FOP0 and FOP1 will be updated. The fault handler can inspect these registers to determine how to proceed.

There are two fault vectors, one for each of the message priorities, stored at known locations in the lower part of the MDP memory. Each vector contains 32 slots that are indexed by the fault number associated with the fault. These vectors must be set as a part of the general system initialization.

2.11.1 Asynchronous Faults

There are two fault-types that occur asynchronously to the execution of the instruction stream; external interrupts, and queue overflow traps. The MDP contains a simple diagnostic port that can be used to read and write the memory of the node, to start and stop the processor core, and so on. One of the commands causes the interrupt fault to be generated at the end of the current instruction.

A queue-overflow is signalled when a message arrives but the target message queue is already full. Handling of this fault may be delayed if a switch to the required priority is disabled. In this case, the network backs up until it can be serviced. The associated fault handler will move a portion of the queue to external memory and then restart the current message.

Care must be taken when writing programs to account for such possible asynchronous faults. As a minimum such a fault will alter the FIP. It is possible to disable such faults during critical regions of code

by setting the Interrupt Bit, the I-bit, or the Fault Bit, the f-bit, appropriately. These flags are discussed more fully later.

2.11.2 Calls

The MDP also provide a simple call mechanism that is based closely on the fault mechanism. The call instruction takes a single operand, an integer, and uses it as an index into a table of 64 possible calls. The IP is saved in the FIP as for a fault.

2.11.3 Unchecked Mode

To increase the flexibility of the MDP and to increase the efficiency of certain sequences of low-level code, it is possible to disable most of the checking performed by the MDP. The major effect of executing in unchecked mode is that the types of operands will be ignored. This mode is controlled by the Unchecked Flag, or U-bit, contained in the current instruction pointer.

2.12 NETWORK INTERFACE

The Message Driven Processor contains a high-performance 3 dimensional router that functions independently of the processor core. Messages are transmitted through the current node or added to the node's message queues without interrupting the integer unit. The programmer is required to perform certain initialisation tasks to configure the message queues, and must be prepared to handle four possible fault conditions as a result of processing messages.

2.12.1 Message Queues

Incoming messages are queued in message queues before being dispatched and processed. There are two message queues, one for each priority level. Each message queue is defined by two registers: the QBM, queue base/mask register, and the QHL, or queue head/length register. The queue base/mask register defines the absolute position and length of the queue in memory. In order to simplify the hardware, the length must be a power of 2, and the queue must start at an address that is a multiple of the length. The queue head/length register specifies which portion of the queue contains messages that have been queued but not processed yet (including the message not yet dequeued by SUSPEND). To avoid having to copy memory, the queue wraps around; if a twenty-word message has arrived and only eight words are left until the end of the queue, the first eight words of the message are stored until the end of the queue, and the next twelve are stored at the beginning. The queue head/length register contains the head and length of the queue instead of the head and tail to simplify the bounds-checking hardware involved in checking user program references to the queue. Below is a diagram of a queue with one message being processed, one more waiting, and a third one arriving.

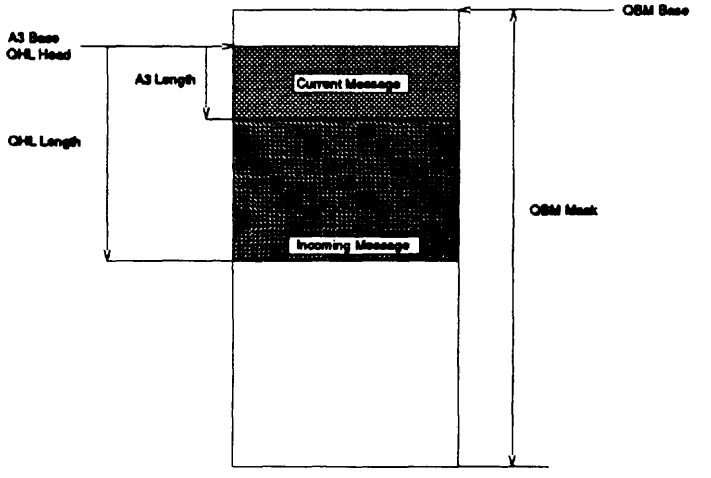


Figure 2.1: A Message Queue.

To reduce contention for the shared on-chip memory and to simplify certain aspects of the queue management hardware, incoming messages are buffered within the network interface in the Queue Row Buffers and written to the memory in blocks of four words. This can cause one, two, or three words to be wasted between messages in the queue. This alignment is transparent to the software; the length and head in QHL are automatically aligned to multiples of four words by the hardware. The length field of the message header specifies the exact length of the message. The effect of this buffering is sometimes visible to users performing low-level debugging of the contents of the message queue, or when writing certain low-level system utilities.

2.12.2 Message Reception

There are two stages in processing of messages: queueing and execution. A message cannot be queued if the D-bit of the associated QBM register is set. This bit is set during reset to allow proper system initialization and may be set by certain kernel system routines. If the D-bit is set, the message will back up into the network but no data will be lost. A queue-overflow trap is requested whenever data arrives and the queue is already full. It will be handled as soon as the priority is runnable.

If the processor is currently executing at a lower priority than the new message and interrupts are enabled, then the message will be dispatched as soon as the first four words are delivered. The A3 register is written with the base field from the QHL and the length field from the bottom 10 bits of the message header. The Q bit in the status register is set to allow accesses to messages that are "wrapped around", such as the twenty-word message in the example above.

The B flag is cleared, P is set to the priority at which the message arrived on the network. If the first word of the message is tagged MSG, then the IP offset and the F and U flags are loaded from the first word of the message, otherwise a message-fault is signalled. The A0-Absolute bit is set.

2.12.3 Suspend

The SUSPEND instruction terminates the processing of the message. First it flushes one message from the proper input queue. Then, if another message (of either priority) is ready, it is executed as described in the Message Reception section. Otherwise, the processor resumes execution of the background priority. A SUSPEND executed in background mode produces indeterminate results.

Note that every SUSPEND corresponds to exactly one message arrival. This SUSPEND terminates the processing of the message and also flushes the message. Therefore, every MDP routine that gets executed by a message must terminate with a SUSPEND at some point.

2.12.4 Message Transmission

The SEND, SEND2, SENDE, and SEND2E instructions are used to send messages. The first word sent specifies the absolute node number of the destination node (i.e. the destination node's NNR value) in the low 16 bits. The high 16 bits are ignored. The type is also ignored so long as it is not CFUT in checked mode. The third argument of each SEND instruction determines the priority at which the message is to be sent over the network: 0 means priority level 0 and 1 means level 1. The priority of the message is independent of the priority of the process that is sending it.

The initial routing word is followed by a number of words which the network delivers verbatim to the destination node. The network does not examine the contents of these words other than to verify that they are not of type CFUT if the processor is in checked mode. The message is terminated by a SENDE or SEND2E instruction, which sends the last one or two words, and tells the network interface that the message is complete. The first word that arrives at the destination node (the second word actually sent, since the routing word is only used by the network and is stripped off during routing) must be tagged MSG.

There are several issues to be aware of when sending messages. The total time between the first SEND and the SENDE should be as short as possible to avoid blocking the network. The user should avoid performing significant computation or taking faults during a sequence of SEND operations. A more subtle error can occur in programs that use both priority 0 and priority 1. If a dispatch to priority 1 were to occur while a thread at priority 0 is in the middle of sending a message, and if that priority 1 thread sends a message, then the messages will be concatenated. The solution to this is to explicitly disable interrupts, by setting the Interrupt Flag, before the priority 0 task begins sending the message.

2.12.5 Faults Associated with the Network

There are four possible faults that can be taken as a result of the use of messages: send-fault, message-fault, early-fault, queue-overflow-fault.

2.12.5.1 Send Fault

This fault occurs when a send instruction is unable to be executed because the network output buffers are full. This is normally the result of a temporary congestion problem in the vicinity of the node that can be expected to clear again relatively quickly. The classic fault handler pauses, backs up the instruction pointer, and retries the send. The network can be backed up for considerable periods of time if a receiver disables one of its message queues, such as when handling queue overflow faults. If a node disables its priority 1 queue, it will not only backup priority 1 messages but can also halt the reception of priority zero messages if an earlier priority 1 message has filled the input buffers. A sophisticated retry handler will include a timeout feature that detects when the network has been blocked for "too long". This fault occurs regardless of the state of the unchecked flag.

2.12.5.2 Message Fault

This fault occurs at the destination node if the header word of the message is not of type MSG. It can occur as the result of a programming error, as a deliberate effort to handle certain messages specially, or because of bit errors in the network media.

One common programming error that can cause an unexpected message fault is to clear the I-bit to accept interrupts without initialising the QHL and QBM registers of both priorities. The MDP constantly inspects these registers whenever the I-bit is clear and will mistakenly conclude that a message is available if they are not set properly. This error usually occurs in simple test programs that did not expect to send or receive messages but wanted to accept external interrupts.

2.12.5.3 Early Fault

The network interface adds words to the message queue four words at a time, and the processor will dispatch as soon as the first four words have been delivered. If a longer message occurs and the code attempts to read an argument that has not been delivered or attempts to suspend before the entire message has arrived, an early fault occurs. This is handled in the same manner as for a send-fault; pause and then retry the instruction. This fault cannot be disabled.

2.12.5.4 Queue Overflow Fault

Queue overflow interrupts are signalled when the last empty word of the queue is written, but may cause an interrupt only when running at the same priority as the queue which overflowed. In other words, if the priority 0 queue overflows and a priority 1 process is currently running then the handler for the queue overflow must wait until all pending priority 1 processes have suspended and the processor has returned to priority 0 before the fault handler will be executed. Likewise, if the priority 0 queue overflows and a background mode process is currently running with interrupts disabled then the handler must wait until the background permits the processor to switch to priority 0.

Note that a queue overflow occurs whenever the message queue contains 1 more word that the mask field of the QBM register for the associated priority. If the queue has the maximum length of $400 words, then the mask field of the QBM is $3FF, ten bits all set to 1, but the length field of the QHL at the time of the fault will have just overflowed from $3FC to $000. This is the correct operation of the QHL register.

This fault may be handled by copying the contents of the queue into memory and then arranging to sequence the messages from there. Writing this code is not for the faint of heart but it is practical. This fault is disabled whenever the I-bit is set or when the F-bit is set. The queue-overflow fault handler must explicitly write to the associate QBM register before returning in order to clear the fault indication.

2.12.6 Initializing the Network

There are several steps that must be followed to initialise the network: the queue registers must be set, the NNR register must be set, and the internal datapath of the routers must be set.

The first step is to allocate the message queues, clear the D-bit in the QBMs, and then accept interrupts by clearing the I-bit.

Every node must be assigned a unique node number and this number must be written to the Node Number Register before any messages can be sent. The nodes of the J-Machine are physically connected in a 3 dimensional space but the MDP cannot trivially determine where is is in this space. There are two standard approaches to informing each node of its node number: write the node-id into a known memory location on each processor from the console before starting the machine, or send a node initialisation message to each processor. The disadvantage to the first approach is that it requires the front-end host to cycle among all of

the nodes and write a different number into every processor. This is potentially quite slow on a 1024 node J-Machine and lacks finesse.

An alternative solution, and one that is demonstrated in one of the programming examples, is to send a message to perform initialisation. This message contains the node-id of the processor as one argument and the maximum node-id of any node in the J-Machine as a second argument. The node sets its own NNR and then determines which of its forward neighbors are present. It computes the node-id for each of the possible dimensions and forwards the initialisation message. This message fans out in wave through the machine. This approach works because the MDP routers always accept the first message after reset.

The MDP contains three routers, one for each dimension, that handle the actual transmission of data. Each router contains an internal address register that cannot be written directly from the integer core. Instead these registers are programmed by sending a message with the correct address in it from the node itself. Thus, when the node determines its node-id and has set the NNR, it must send a message to itself at each priority. This message can immediately suspend as it was sent only for this effect. This is also demonstrated in the programming example.

2.13 EXTERNAL MEMORY INTERFACE

The external memory interface supports the use of up to 1 Mword of off-chip DRAM. It provides single-bit error correction and double-bit and multiple nibble error detection, DRAM refresh, and support both static-column and page-mode memory devices. It is controlled by two registers: the refresh timer, and the error control.

DRAM refresh is disabled after reset. It is enabled by writing a value to the 8 bit emi_rtc register. This register is memory mapped at location $FFFF0. The value stored into this location is loaded into an internal counter and incremented by 1 on every cycle. When this value is incremented to $FF a DRAM refresh is performed and the start count is reloaded into the counter.

The error control register, emi_erc at location $FFFF1, controls the error correction mode, the type of DRAM being used, and counts the number of single-bit errors corrected.

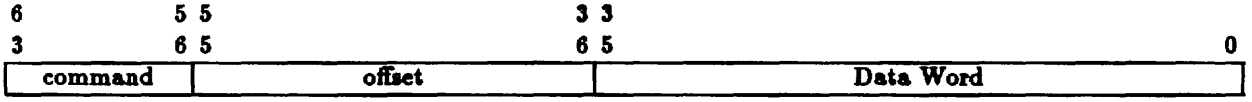
The M-bit indicates the type of DRAM being supported. It is set for page-mode and clear for static-column. Setting the D-bit disables error correction. The M-bit is unspecified after reset and the D-bit is set.

The memory interface includes a 12 bit data bus. It takes at least 4 cycles to read and write external memory, one for setup and three more to transfer an MDP word. If error-correction is enabled then an additional cycle is performed to read or write the error syndrome. No additional delay is incurred to perform the error check itself unless the data being read must be corrected. A dramerr trap is taken if the data read contains more than 1 bit error.

2.14 DIAGNOSTIC INTERFACE

The diagnostic interface is used to "boot" the J-Machine and to support low-level debugging operations. It can be used as the basis for all interactions between the machine and the front-end host but the intent is that it be used to install the initial code and that a network interface adapter be used after that.

The MDP contains an internal 64-bit shift-register that is used to enter commands to read and write memory, to stop, start, and step the processor, and to cause an external interrupt. The format of the register is shown below:



This register is loaded through a bit-serial shift-port that is controlled by a special host-interface board present in the front-end, and by registers on each processor board. When the J-Machine is first turned on and reset, the memory is in an unknown state and only a few of the registers have a defined state. The processor core is halted, i.e. it is not fetching instructions. Of particular concern is that the network control registers have not been initialised and therefore the MDP will not accept messages. The diagnostic port is used to install, at least, the initialisation code for the programming system and to start the processor at $400 in a0-absolute mode. This code must initialise the major state and enable interrupts. It is also possible to generate an asynchronous interrupt fault through this interface.

Chapter 3

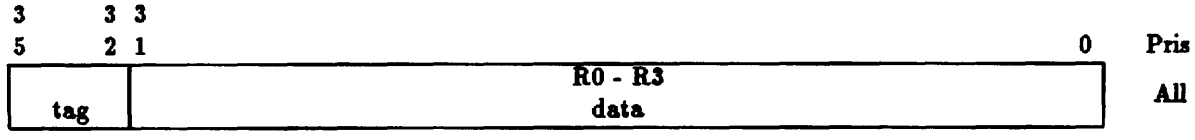
REGISTERS AND MEMORY

The MDP register file is divided into several different classes based on the associated functionality; there are a small number of general purpose registers that can hold values of any type, a set of address registers, ID registers used for name translation, and a number of special purpose registers used to support fast trapping and, the network interface, and so on. The primary programmer registers are replicated for each priority of operation to support fast context switching among the background and the two message priorities.

The figures included below indicate which priorities are supported on the right. For example, there is a unique set of general purpose registers for every priority, the ID registers are replicated for priority 0 and priority 1 but not the background, and the NNR is a "global" register and is not replicated for any of the priorities.

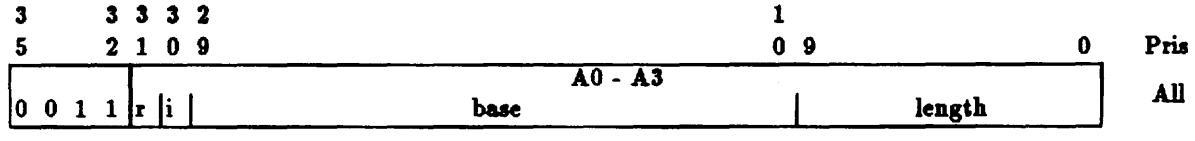
3.1 DATA REGISTERS

There are four 36 bit data registers for each of the three priorities and each register can store values of any type. No special meaning is assigned to any of the fields of these registers. The core arithmetic and logical operations require one operand to be in a data register and for the destination to be a data register. Long constants, described in more detail later, are always stored in R0 making this register slightly less valuable for storing intermediate values than the other registers. Instruction formats and restrictions are discussed in more detail in a later section.



3.2 ADDRESS REGISTERS

The address registers A0-A3 are used for all memory references. As for the data registers, there are three independent sets of address registers. These registers are always read as ADDR-typed values regardless of the type of value written to them. Two of the registers, A0 and A3, have special significance during program execution (see a0-absolute/relative mode and a3-relative mode).



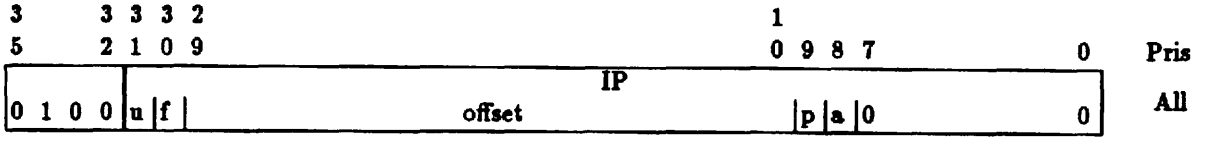
Setting the invalid bit causes all memory references using the address register to signal an invalid address, INVADR, fault. This bit may be set by certain runtime system primitives to assist in memory compaction, object relocation, and so on.

Setting the relocatable bit indicates that the address refers to an object that may be moved. This bit is purely for the convenience of the runtime system and has no effect on the processing performed by the MDP. This bit allows a post-heap-compaction invalidation of only the relocatable addresses, leaving the locked down physical addresses intact.

Note carefully that the MDP is a word-addressed machine; the offset stored in an address register is to the start of a full 36-bit word within the memory space of the processor. This representation permits the addressing of 1 Mword, effectively 4 Mbytes, of tagged data.

3.3 INSTRUCTION POINTER

The instruction pointer indicates the next instruction to be fetched and contains a number of flags that have a significant impact upon the execution of the program.



The offset stored in the IP is either an absolute address within the memory space of the MDP or an offset relative to the base of the current method pointed to by A0 (see A0-Absolute Addressing). The current mode is indicated by bit 8, the A0-Absolute bit. When this bit is set, all memory references via A0 are absolute addresses and the actual value in A0 is ignored. When it is clear, they are relative to the value in A0.

Two instructions may be stored in a single word. The phase bit, bit 9, indicates which instruction is being fetched. If P is 0, the first instruction, stored in the upper half of the word, is fetched otherwise the instruction stored in the low half is executed.

The U-bit, bit 31, is a copy of the unchecked bit stored in the status register. Changing it by altering the IP register changes it in the status registers and vice-versa.

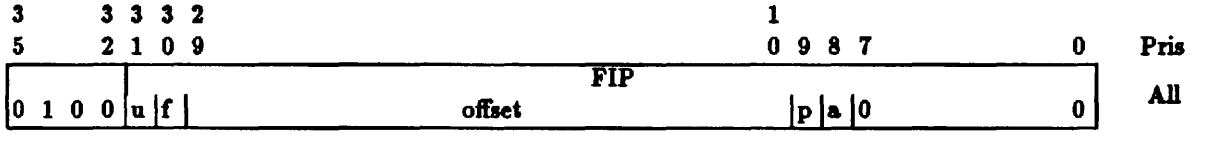
The F-bit, bit 30, is similarly a copy of the fault bit of the status register. Its use is discussed later.

3.4 FAULT REGISTERS

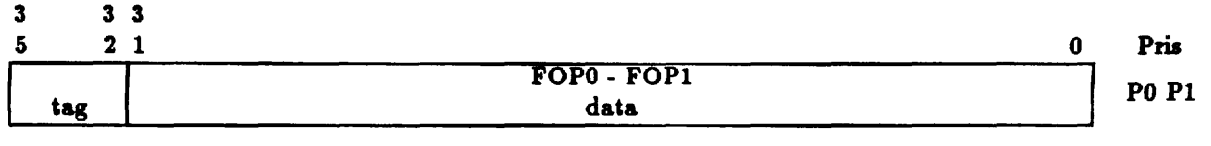
A key design feature of the MDP is hardware support for trap handling in exceptional circumstances. Traps are taken when data type tags differ from the expected value, to support synchronisation mechanisms via the CFUT type, to assist in the management of the message queue, and so on. A complete description of the fault types is included later. The call mechanism is based closely on the fault mechanism.

When an exceptional situation is detected, up to four registers are updated with the key processor state to permit the software trap handler to determine the reason for the fault.

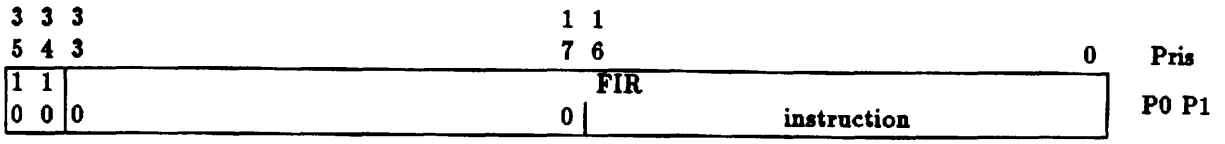
The Faulted Instruction Pointer, FIP, is loaded with the current IP when the fault is taken. Since the IP is pre-incremented, the FIP contains the IP of the instruction immediately following the faulted instruction. If the instruction must be reexecuted after taking some appropriate action, the fault handler is responsible for backing up the IP in software.



The faulted instruction registers, FOP0 and FOP1, are loaded with the values of the Op0 and Op1 operands whenever an instruction-specific fault occurs. Note that it is necessary to have a detailed understanding of the instruction format to determine which value will be stored in each register. If a fault occurs that was not caused by a specific instruction, then the value written into these registers is indeterminate. If a faulting instruction has no Op0 or Op1 operand, then the value of FOP0 and/or FOP1 is indeterminate.

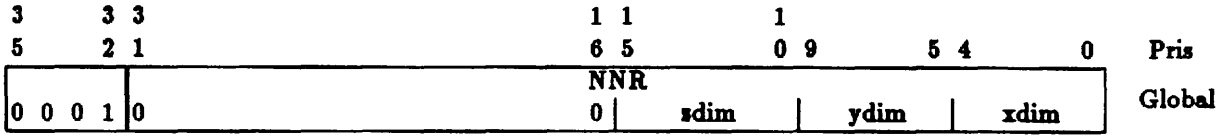


The Faulted Instruction Register, FIR, contains the instruction that caused the exception or NIL if the fault is not associated with any instruction. The instruction is always stored in the low 17 bits regardless of which half of the word it appeared in during the program execution.



3.5 NETWORK CONTROL REGISTERS

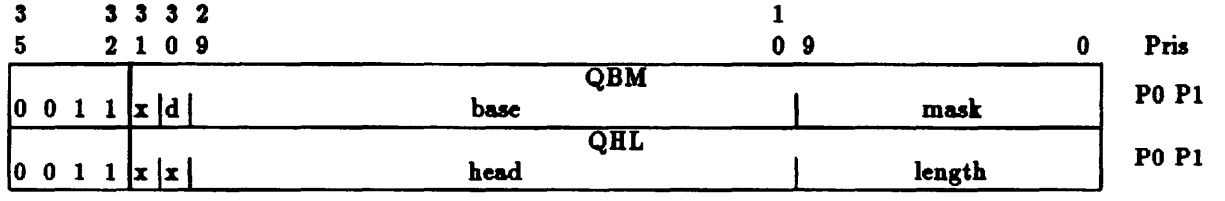
The node number of a node and the extent of the two message queues must be set by the programmer during program initialization.



The node number register, NNR, contains the network node number of this node. It consists of an X field, a Y field and a Z field indicating the position of the node in the 3D network grid. Its value identifies the processor in the network and is used for routing. The NNR should be initialised by software after a reset and left in that state. The NNR is read and written as an INT-tagged value.

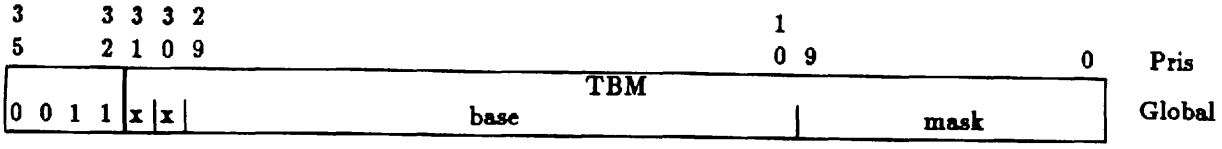
The queue base/mask register, QBM, indicates the base and extent of the input message queue. The base is the first memory location used by the queue. The mask must be of the form $2^n - 1$, with n >= 2. The size allocated to the queue is equal to the mask plus 1. There is one more restriction: (base XOR mask) = 0 must hold. This effectively means that the base must be a multiple of the size of the queue, and this size must be a power of 2. These conditions allow queue access and wraparound to work by simply ANDing the offset within the queue with the mask and then ORing with the base. The disable bit, bit 30, should normally be zero. Setting it disables message reception at the priority level of the QBM register, which may cause messages to be backed up in the network. This should be done only under special circumstances, such as when the queues are being moved. It is set after reset. The QBM register is read and written as an ADDR-tagged value.

The queue head/length register, QHL, contains two fields, head and length that describe the current dynamic state of the queue. Head is an absolute pointer (i.e. relative to the beginning of memory, not the beginning of the queue) to the first word that contains valid data in the queue, while length contains the number of valid data words in the queue. The length is zero when the queue is empty, and 1 greater than the mask when the queue is full. The length can also be zero if the message queue is the maximum possible length, 1024 words, and the queue is full. The queue is clearly full if the length is zero and yet the queue-overflow fault has been signalled. QHL is read and written as an ADDR-tagged value.



3.6 NAME CACHE CONTROL

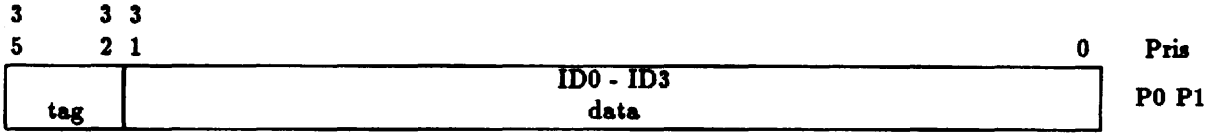
The translation base/mask register, TBM, is used to specify the location of the two-way set-associative lookup table used by the XLATE and ENTER instructions. The format of the TBM register is similar to that of the QBM register. Again, base is the first memory location used by the table. The mask must be of the form $2^n - 1$, with n >= 2. The number of words occupied by the table is equal to the mask plus 1. As in QBM, (base XOR mask) = 0 must hold. TBM is read and written as an ADDR-tagged value.



3.7 ID REGISTERS

The ID registers provide support for the maintenance of a global namespace within the J-Machine for certain programming systems. In such systems objects will be named using an ID, or handle, and these IDs will be the sole reference communicated between methods distributed across the machine. This design permits objects to be relocated within the private memory of a given node or to be transferred to another node without the requirement that a very large number of physical addresses be invalidated within the machine. Operating system primitives provide the primary mechanisms for translating these handles into a physical address at the site of the object.

These name translations, from a name to an ADDR-tagged value to be stored in an address register, can be accelerated using the enter and translate operating provided by the name cache system. In normal practice, ID register N will hold the ID of the relocatable object pointed to by address register N. The name translate operator automatically stores the ID in the appropriate register during an XLATE.

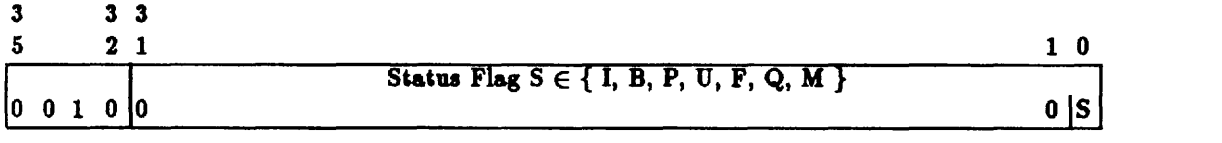


3.8 MEMORY ADDRESS REGISTER

The Memory Address Register, MAR, is provided for debugging purposes and is probably of little use to the applications programmer. This register contains the absolute address of the last memory location read or written by the execution of an instruction. The MAR can only be read.

3.9 PROCESSOR STATUS FLAGS

The status register is a collection of flags that may be accessed individually using READR, WRITER, or the alias MOVE. The status register cannot be accessed as a unit. It contains these flags:



I:	Interrupt Mask	0: Interrupts Allowed	1: Interrupts Disabled
B:	Background Execution	0: Message	1: Background
P:	Priority Level	0: Level 0	1: Level 1
U:	Unchecked Mode	0: Checked	1: Unchecked
F:	Faults Disabled	0: Normal	1: Faults Disallowed
Q:	Queue Wrap Around	0: A3 Points Into Memory	1: A3 Points Into Queue
M:	Message Flag	0: Message Send Complete	1: Message Being Sent

Note carefully that the F and U flags are mirrored in the current instruction pointer. They are set or cleared explicitly by the user or as a result of loading the instruction pointer from a fault vector, from the call vector, from the header of a message, or by the LDIP and LDIPR instructions.

The priority and background flags specify the current priority level of execution. The highest level is priority 1, with the settings P=1, B=0. Below that is priority 0, with P=0, B=0. The lowest priority level is background, with B=1.

The interrupt mask flag determines whether the current process may be interrupted. Setting this flag disables all interrupts. Clearing this flag allows interrupts. There are two types of interrupts: synchronous faults associated with the current instruction stream, and asynchronous faults which are not. The synchronous faults are disabled whenever the I-bit is set or when the MDP is sending a message to priority 1 (see M-bit below). The two asynchronous faults, message queue overflow and external interrupt, are also disabled when the F-bit (below) is set.

The fault flag is used to identify periods during which it is not acceptable to take a fault. It is contained in the status register and is mirrored in the current instruction pointer. It is normally set while entering a fault handler, as part of the IP stored in the fault vector. It can be cleared explicitly once the fault registers are saved in memory or when the original instruction pointer is reloaded from the FIP register at the end of the fault handler. If a fault occurs when this bit is set, the fault is ignored and a CATASTROPHE fault is taken instead. The handler for this fault will generally save the state of the processor and then signal the user to indicate that a programming error has occurred. The fault flag is also used to disable the asynchronous queue overflow and external interrupt faults.

The unchecked mode flag determines whether TYPE, CFUT, FUT, TAG8, TAG9, TAGA, TAGB, and OVERFLOW faults are taken; when this flag is set, these faults are ignored, which allows more freedom in manipulation of data but provides less type checking. There is also a copy of this flag in the IP register. Changing this flag changes it in the IP register and vice versa. As with the F flag, there are three copies of the U flag, one for each priority level and one for background mode.

The A3-Queue bit, when set, causes A3 to "wrap around" the appropriate priority queue. This is included to allow A3 to act transparently as a pointer to a message, whether it is still in the queue, or copied into the heap. If the message is still in the queue, then setting the Q bit allows references through A3 to read the message sequentially, even if it wraps around the queue. If the message is copied into an object, then leaving the Q bit clear allows normal access of the message in the object. The Q bit is set on message dispatch, and it is left to the software to clear the Q bit when a message is copied into the heap. Either way, the access of the message pointed to by A3 looks like any other reference through an address register and bounds checking is performed. Note that when the Q bit is set, the head of QHL should point to the same place as the base of A3 (since the start of the queue is also the start of the next message to be processed). There is a Q bit for each priority level, but no Q bit for background mode (because there is no queue for background mode).

The two M-bits indicate whether a message is being sent TO a particular priority. MP0 is set if the message is being sent to priority zero and MP1 is set if a message is being sent to priority one regardless of the current priority. These bits are set by the use of the SEND instruction and cleared by SENDE. MP0 is set for the support of certain system oriented routines but has no effect on the execution of the processor. The MP1 bit disables interrupts i.e. it is not possible for the MDP to process an asynchronous fault or dispatch a new message while it is sending a P1 message.

3.9.1 Background Priority

It has been noted that the background priority is selected when B=1. When B=1 and one of the standard background registers is read or written, the appropriate background register will be accessed regardless of the P bit. However, it is possible to access a non-background register, e.g. QHL, directly with B=1. In this case, the register that is selected is determined by the current value of P. If P=0, the QHLP0 will be accessed otherwise QHLP1 is accessed. The P flag is set to 0 during chip reset and we use this as the definition of background i.e. B=1 and P=0. It is generally unwise to run with B=1 and P=1 although there may be reasons to do so in special situations.

3.10 MEMORY MAP

The MDP contains 4K words of on-chip SRAM. The prototype J-Machine adds a further 252K words of error-corrected off-chip DRAM for a total of 256K words. Certain memory locations have special purposes assigned to them by the hardware. These are outlined in the table below.

FROM	TO	USE
$00000	$0003F	Priority switchable memory 0
$00040	$0007F	Priority switchable memory 1
$00080	$0009F	Priority 0 Fault Vectors
$000A0	$000BF	Priority 1 Fault Vectors
$000C0	$00100	Call table
$00100	$00FFF	Uncommitted on-chip RAM
$01000	$040FF	External memory on J-Machine prototype
$04100	$FFFEF	Additional external Memory Address space
$FFFF0	$FFFF1	EMI control registers

Within the uncommitted internal RAM, the operating system will allocate several hundred words to the message queues and the XLATE cache and leaves the rest of RAM for user programs. The call vector table length is operating system definable, but its base must be location $000C0.

3.10.1 Priority Switchable Memory

In order to allow each priority level to have 32 private temporaries, the first 64 words of memory are decoded specially. When accessing one of these 64 words, the current state of the P flag is XORed with bit 5 of the address; hence, referencing location 1 accesses physical location 1 when running in priority level 0 (P flag clear) or location 33 when running in priority level 1 (P flag set). This scheme lets the operating system and user programs use memory locations 0 through 31 as temporaries private to the current priority level. The other priority level's temporary "globals" can be accessed as locations 32 through 63.

3.11 EXCEPTIONS

3.11.1 Reset

When the processor is reset, the status register flags are set as follows: Q*=0, U*=1,F*=0, M*=0, I=1, B=1, P=0. The A bit in the background IP and D bits in both QBM registers are set. The background IP offset is set to $400. The program that gets executed after a reset should set up the queues, NNR, and at least some of the fault vectors and then clear the I flag and the D bits in the QBM registers to allow message reception.

3.11.2 Fault Processing

When a fault occurs, the instruction that caused the fault is saved in the FIR register, the current IP (which points one instruction beyond the faulting instruction) is saved in the FIP register, and the values of the Op0 and Op1 operands (if any) are saved in the FOP0 and FOP1 registers; the IP is then fetched from the memory location whose address is equal to the fault number plus the base of the fault vector table of the current priority (when in Background mode the Priority flag is used to select the fault vector). If the F bit is set and a fault occurs then the IP is loaded from the CATASTROPHE fault vector. The U, A, and F bits of the IP that gets loaded may change the processor state. U determines if this priority is in unchecked mode, A determines if A0 absolute mode is in effect, and the F bit determines whether the fault is non-reentrant and interruptible.

3.11.3 System Calls

A system call (via the CALL instruction) mimics some of the behavior of a fault to provide convenient access to system routines. When a CALL occurs, the base of the system CALL vector table is added to the CALL operand, and the contents of this location are fetched, yielding a call handler IP. The current IP (which points to the next instruction) is saved in the current priority's FIP register. Execution then begins by loading the call handler IP (which sets the F, A, and U bits in the status register to the values in the call handler IP).

3.11.4 Interrupts

There are three types of interrupts supported on the MDP: priority switches, queue overflow interrupts, and external interrupts. Priority switches may occur at any time, provided that the I and MP1 flags are both clear. Queue overflow and external interrupts may only occur when all of I, MP1, and F flags are clear. Priority switches should be the most common interrupts; these occur when a message arrives in the queue of

a priority higher than the current priority. Thus, priority 1 messages can interrupt priority 0 or background mode, and priority 0 messages can interrupt background mode. The handler for a priority switch is the interrupting message itself.

Queue overflow faults are discussed in the section on the network interface.

External interrupts are similar to queue overflow interrupts except that whenever the I, MP1, and F flags are clear and an external interrupt is signalled, a fault is signalled at the current priority and the IP is loaded from the INTERRUPT fault vector. The interrupt is handled as a process of the same priority as the priority which it interrupted. An external interrupt is signalled by an external interrupt pin on the MDP package.

Interrupts may occur only between instructions. After an interrupt the FIP points to the next instruction of the interrupted sequence.

3.11.5 Fault Types

Name	Number	Description
CATASTROPHE	$0	Double fault,bad vector, or other catastrophe.
INTERRUPT	$1	Interrupt signalled by diagnostic interface.
QUEUE	$2	Message queue about to overflow.
SEND	$3	Send buffer full.
ILGINST	$4	Illegal instruction.
DRAMERR	$5	Double bit error in the external RAM.
INVADR	$6	Attempt to access data through address register with I bit set.
ADRTYPE	$7	The address index is not an integer.
LIMIT	$8	Attempt to access object data past limit.
EARLY	$9	Attempt to access data in message queue before it arrived.
MSG	$A	Bad message header.
XLATE	$B	XLATE missed.
OVERFLOW	$C	Integer arithmetic overflow.
CFUT	$D	Attempted operation on a word tagged CFUT.
FUT	$E	Attempted operation on a word tagged FUT.
TAG8	$F	Attempted operation on a word tagged TAG8.
TAG9	$10	Attempted operation on a word tagged TAG9.
TAGA	$11	Attempted operation on a word tagged TAGA.
TAGB	$12	Attempted operation on a word tagged TAGB.
TYPE	$13	Operand(s) with a bad tag type used in an instruction.
	$14-$1F	Reserved for future faults.

Note: If multiple faults occur simultaneously the fault vector chosen is the one that has the highest precedence. Each fault is assigned a precedence by its fault number; lower fault numbers correspond to higher precedence.

Chapter 4

INSTRUCTION SET

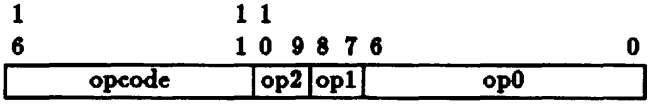
The instruction set of the MDP has been designed to be convenient, flexible, and to permit a reasonably tight encoding to make efficient use of memory. It is not a true RISC-like load/store instruction set, but it is certainly streamlined and should be readily familiar to users of current microprocessors. The processor core supports three-operand addressing, two for the source and one for the destination, and the basic instructions operate primarily on the general registers. In contrast to common convention in the current generation of so-called RISC chips, one of the operands can reference a slot in memory. There is one addressing mode; base with offset. The offset can be a small immediate or the contents of a general register. There is also support for accessing a small set of common program constants in a more efficient manner than the general mechanism for long constants. In the description that follows, the assembler syntax for the standard assembler, MDPSim, is included for clarity.

4.1 INSTRUCTION ENCODING AND ADDRESS MODES

The program executed by the MDP consists of instructions and constants. A constant is any word not tagged INST0 through INST3 that is encountered in the instruction stream. When a constant word is encountered, that word is loaded into R0 and execution proceeds with the next word.

Every instruction is 17 bits long. Two 17-bit instructions are packed into a word. Since a word has only 32 data bits, two tag bits are also used to specify the instructions. The instruction in the high part of the word is executed first, followed by the instruction in the low part of the word. As a matter of convention, if only one instruction is present in a word, it should be placed in the high part, and the low part of the word set to all zeros.

The format of an instruction is as follows:



The opcode field specifies one of 64 possible instructions. The other fields specify three operands; instructions that don't require three operands may ignore some of the operand fields. Operands 1 and 2 must be data registers; their numbers (0 through 3) are encoded in the 1st reg # and 2nd reg # fields. Operand 2, if used, is always the destination of an operation and operand 1, if used, is always a source. Op0 is encoded in one of two ways depending on the operator. It can be used as a source or the destination and supports a variety of accesses.

4.1.1 Normal Addressing Mode

This is the addressing mode used for Op0 by almost all of the MDP's instructions. It permits access to the data and address registers, the use of short constants and of certain long constants, and for address-register indirect accesses. The operand can be a source or a destination, but it is nearly always the source. Typical uses of this operand are:

```
add     r0, r1, r2          ; r1 is Op0
add     r0,  3, r0          ; a small immediate in the range -16 to 15
add     r2, [5, a2], r2     ; Offsets in the range 0 to 15
and     r0, $3FF, r0        ; A special immediate constant
add     r0, a1, r3          ; Uncommon. Only permitted in unchecked mode

send2   [3, a2], r1, 0      ; Op0 is sent first, therefore unusual syntax

move    27, r0              ; Immediates in the range -32 to 31
move    [49, a1], r2        ; Offsets in the range 0 to 63
```

Primitives with two source operands can represent the immediate integers -16 to 15, and immediate offsets in the range 0 to 31. If there is only one source, then the second operand field is used to extend the range by 2 bits, i.e. -32 to 31 for values, and offsets in the range 0 to 63. If the instruction has just one source operand then the extension is always taken from Op2 otherwise it is instruction dependent. The complete specification of this addressing mode is indicated below:

6						0	Syntax	Addressing Mode
Normal Addressing Mode								
0	0	0	0	0	Rn		Rn	Data Register Rn
0	0	0	0	1	An		An	Address Register An
0	0	0	1	0	0	0	NIL	Immediate Constant NIL (SYM:0)
0	0	0	1	0	0	1	FALSE	Immediate Constant FALSE (BOOL:0)
0	0	0	1	0	1	0	TRUE	Immediate Constant TRUE (BOOL:1)
0	0	0	1	0	0	0	$80000000	Immediate Constant INT:$80000000
0	0	0	1	1	0	0	$FF	Immediate Constant INT:$000000FF
0	0	0	1	1	0	1	$3FF	Immediate Constant INT:$000003FF
0	0	0	1	1	1	0	$FFFF	Immediate Constant INT:$0000FFFF
0	0	0	1	1	1	1	$FFFFF	Immediate Constant INT:$000FFFFF
0	0	1	Rx		An		[Rx, An]	Offset Rx in object An
0	1	imm					imm	Immediate (signed)
1	imm				An		[imm, An]	Offset imm (unsigned) in object An

4.1.2 Register Oriented Addressing Mode

The register-oriented op0 mode is used instead of normal op0 mode by the READR, WRITER, and LDIPR instructions. This mode provides access to registers at different priorities and to the special registers. The register-oriented op0 mode encodings are as follows:

6						0
Register Addressing Mode						
B	P	0	0	0	Rn	
B	P	0	0	1	An	
-	P	0	1	0	IDn	
B	P	0	1	1	0	0
-	P	0	1	1	0	1
-	P	0	1	1	1	0
-	P	0	1	1	1	1
-	P	1	0	0	0	0
-	P	1	0	0	0	1
B	P	1	0	0	1	0
-	-	1	0	0	1	1
-	-	1	0	1	0	0
-	-	1	0	1	0	1
-	-	1	0	1	1	0
-	-	1	0	1	1	1
-	-	1	1	0	0	0
-	-	1	1	0	0	1
-	-	1	1	0	1	0
B	P	1	1	0	1	1
B	P	1	1	1	0	0
-	P	1	1	1	0	1
-	-	1	1	1	1	0
-	-	1	1	1	1	1

Syntax	Addressing Mode
Rn	Data Register Rn
An	Address Register An
IDn	ID Register IDn
FIP	Faulted Instruction Pointer
FIR	Faulted Instruction Register
FOP0	Faulted Operand Zero Register
FOP1	Faulted Operand One Register
QBM	Queue Base/Mask Register
QHL	Queue Head/Length Register
IP	Instruction Pointer
TBM	Translation Base/Mask Register
NNR	Node Number Register
MAR	Memory Address Register
	Unused (ILGINST Fault)
	Unused (ILGINST Fault)
P	Priority Level Flag
B	Background Execution Flag
I	Interrupt Flag
F	Fault Flag
U	Unchecked Flag
Q	A3-Queue Flag
M	Send Flag
	Unused (ILGINST Fault)

B represents the use of the Background register set or one of the two priority register sets. The B bit is XORed with the Background Flag and a register set chosen according to the result; 1 indicates the background registers, while 0 indicates the register set chosen by the P bit relative to the present priority. The assembler syntax for specifying a register belonging to the background is the register name followed by a "B".

P represents the priority of the register being accessed, and is relative to the current priority. 0 indicates the current priority, while 1 indicates the other priority. The assembler syntax for specifying a register belonging to the other priority is the register name followed by a backquote (').

4.1.3 Instruction Row Buffer

The message driven processor includes a 4 word Instruction Row Buffer, IRB, to decrease contention for the on-chip SRAM. When instructions are being fetched from on-chip memory, the IRB is loaded with up to 8 instructions in a single cycle and then the instruction decoding occurs from the IRB. The words fetched to fill the IRB are aligned on a four word boundary even if the code branches into the middle of such a block. Branch instructions always refill the IRB even if the target instruction is already in the IRB. The IRB is not used for instructions that are located in off-chip memory.

As with most processors, programmers must be wary of mutating the instruction stream that is about to be executed. Attempting to modify an instruction that has already been fetched into the IRB will not affect the current execution of that code but will successfully modify the in-memory version of the instruction and all future executions of the instruction.

4.2 MOVE AND TYPE OPERATIONS

There are two classes of move operations; the read/write pair that uses the standard addressing mode, and the readr/writer pair that use an extended mode that permits access to the special registers and to the

registers at other priorities (see normal addressing mode and register-oriented addressing mode).

			Source Types
READ	Src, Rd	Rd ← Src.	All but CFut
READR	Src, Rd	Rd ← Src.	All but CFut

In checked mode, these operations fault if the operand is a CFUT.

WRITE	Rs, Dst	Dst ← Rs.	All
WRITER	Rs, Dst	Dst ← Rs.	All

The WRITE instruction may be used to write a value of any type into memory. The WRITER instruction is used to write values to registers. In checked mode, type checking is done for values being moved into an address register or the IP. All types, including CFUT, may be moved into the other registers.

LDIP	Src	IP ← Src.	IP
LDIPR	Src	IP ← Src.	IP

These instructions load the IP using normal addressing mode and register addressing mode respectively. In checked mode, the value must be of type IP.

CHECK	Rs, Src, Rd	Rd ← BOOL:tag(Rs) == Src	All,Int
RTAG	Src, Rd	Rd ← INT:tag(Src)	All but CFut
WTAG	Rs, Src, Rd	Rd ← Src:Rs	All,Int

These operator permit the type of a value to inspected or modified. Rs can be of any type for Check and Wtag. Src must be an integer in checked mode. Src is treated as in integer in the range 0-15 inclusive in unchecked mode. Rtag faults CFUT in checked mode.

4.3 ARITHMETIC OPERATIONS

The MDP implements a full set of arithmetic operations on integers. Integers are represented as signed 32-bit fixnums in 2's complement format. The format of a typical arithmetic instruction is

```
<op>    Rs, Src, Rd
```

This means that Rs and Rd must be one of the four general registers R0-R3. The Src operand is selected from the set of "normal" addressing modes described above. Note that using an address register as a source would only be appropriate in unchecked mode. Similarly, 3 of the special constants are not of type INT and would only be appropriate in unchecked mode.

ADD	Rs, Src, Rd	Rd ← Rs + Src	Int,Int
CARRY	Rs, Src, Rd	Rd ← Carry(Rs + Src)	Int,Int
SUB	Rs, Src, Rd	Rd ← Rs - Src	Int,Int
NEG	Rs, Src	Rd ← -Src	Int

Carry returns 1 if adding the two numbers would generate an unsigned carry and 0 otherwise. It should not be used in checked mode, as it causes an overflow under the same conditions that add overflows. Add and sub produce results modulo 2^{32} in unchecked mode. In unchecked mode the type of Rd is the same as the type of Rs. An overflow occurs in checked mode when the signed result is not the sum/difference of the signed operands. NEG can overflow if Src = $80000000.

MUL	Rs, Src, Rd	Rd ← Low 32 bits of Rs*Src	Int,Int
MULH	Rs, Src, Rd	Rd ← High 32 bits of Rs*Src	Int,Int

These operations generate a 64 bit product of their 32 bit signed inputs. Mulh returns the high 32 bits of this product while Mul returns the low 32 bits. The tag of the result is always the tag of the first operand. In checked mode these operations fault if the type of both source operands is not integer. The mul instruction

faults overflow if the result is not the product of the inputs. The mulh instruction never faults overflow.

ASH	Rs, Src, Rd	Rd ← Rs << Src (arithmetic)	Int,Int
LSH	Rs, Src, Rd	Rd ← Rs << Src (logical)	Int,Int

Src may be negative and may be very large. It is not treated modulo 32; instead, Rs is shifted by Src bits to the left or right if Src is negative, whatever Src happens to be. For example, if Src = -50, Rd is set to 0 by LSH and by ASH when Rs >= 0 and to -1 by ASH when Rs < 0. ASH treats Rs as a signed quantity, while LSH treats it as unsigned. An overflow occurs when Src > 0 and significant bits are shifted from the number; bits shifted to the right from the number are ignored. In unchecked mode the type of Rd is the same as the type of Rs, and Src is treated as if it were a signed integer.

ROT	Rs, Src, Rd	Rd ← Rs rotate left Src	Int,Int

This is a rotate instead of a shift, so bits shifted out of the left side of Rs are shifted back at the right side. Src is an integer treated modulo 32 (since a rotate of 32 bits is the identity transformation). In unchecked mode the type of Rd is the same as the type of Rs.

FFB	Src, Rd	Rd ← Find First Bit	Int

Rd is loaded with an integer value between 0 and 31, inclusive. This indicates how many bits must be traversed, going from left to right starting from bit 30, in order to find the first bit not equal to the sign bit (bit 31). (for example, FFB($80000000)=0, FFB($E0000000)=2, and FFB($20000000)=1) This is useful for normalising floating point values.

4.4 LOGICAL OPERATIONS

The logical operations operate are similar to the arithmetic ones expect they accept either integer or boolean operands in checked mode. A type-fault occurs if the arguments are not of the same type in checked mode.

AND	Rs, Src, Rd	Rd ← Rs AND Src	Int,Int Bool,Bool
NOT	Src, Rd	Rd ← NOT Src	Int Bool
OR	Rs, Src, Rd	Rd ← Rs OR Src	Int,Int Bool,Bool
XOR	Rs, Src, Rd	Rd ← Rs XOR Src	Int,Int Bool,Bool

4.5 COMPARISON OPERATIONS

These operations produce a boolean result based on the comparison of the two source arguments. The addressing modes are the same as for the arithmetic and logical operators.

GE	Rs, Src, Rd	Rd ← Bool:Rs $\geq$ Src	Int,Int Bool,Bool
GT	Rs, Src, Rd	Rd ← Bool:Rs > Src	Int,Int Bool,Bool
LE	Rs, Src, Rd	Rd ← Bool:Rs $\leq$ Src	Int,Int Bool,Bool
LT	Rs, Src, Rd	Rd ← Bool:Rs < Src	Int,Int Bool,Bool

A TYPE fault occurs in checked mode if Rs and Src are not either both integers or both booleans. False is regarded as being less than True.

EQUAL	Rs, Src, Rd	Rd ← Bool:Rs = Src (Data)	Int,Int Bool,Bool Sym,Sym
NEQUAL	Rs, Src, Rd	Rd ← Bool:Rs $\neq$ Src (Data)	Int,Int Bool,Bool Sym,Sym

A TYPE fault occurs in checked mode if Rs and Src are not both integers, both booleans, or both symbols. In unchecked mode the tags are ignored and the data portions are compared.

EQ	Rs, Src, Rd	Rd ← Bool:Rs = Src (Pointer)	All but CFut or Fut
NEQ	Rs, Src, Rd	Rd ← Bool:Rs $\neq$ Src (Pointer)	All but CFut or Fut

Both the data and the tag have to match for two pointers to be EQ. This is different from EQUAL in

unchecked mode. These primitives only fault in checked mode if one of the arguments is either CFUT or FUT.

4.6 BRANCH OPERATIONS

The branch instructions permit conditional and unconditional branches. Because the IP will have already been incremented, an offset of zero will branch to the next instruction. The phase bit of the IP is always cleared. If Src is positive, the branch is forwards otherwise it is backwards. The Src may be a 7 bit immediate displacement, providing a range of -64 to 63 words, or the contents of a general data register.

BR	Src	Branch forward Src words	Int
BZ	Rs, Src	Branch if Data(Rs) = 0	Int,Int
BNZ	Rs, Src	Branch if Data(Rs) $\neq$ 0	Int,Int
BF	Rs, Src	Branch if Bit0(Rs) = 0	Bool,Int
BT	Rs, Src	Branch if Bit0(Rs) = 1	Bool,Int
BNIL	Rs, Src	Branch if Rs = NIL	All but CFut,Int
BNNIL	Rs, Src	Branch if Rs $\neq$ NIL	All but CFut,Int

In unchecked mode, BNIL and BNNIL compare both the tag and data with NIL. BF and BT inspect only the least significant bit of the data portion, while BZ and BNZ compares the full data portion but does not check the tag.

4.7 NETWORK OPERATIONS

SEND	Src, P	Send Src at priority P	All but CFut
SENDE	Src, P	Send Src and terminate	All but CFut
SEND2	Src, Rs, P	Send Src then Rs	All but CFut
SEND2E	Src, Rs, P	Send Src then Rs and terminate	All but CFut

Send one or two words onto the network. When two words are sent, the one from Src is sent before the word in Rs; hence, please note the unusual assembler syntax order of Src and Rs. SENDE and SEND2E indicate the end of the message to the network hardware after the words they send. SEND and SEND2 set the M Flag for the priority to which the message is being sent, while SENDE and SEND2E clear the M Flag. The op2 field is used to encode which message priority to send the message on.

4.8 SPECIAL INSTRUCTIONS

NOP — No Operation

Used to pad instructions when aligning for the target of a branch instruction or to accommodate a long constant.

SUSPEND — Terminate Thread

Pops the current message from the message queue and schedules the next message. Should not be called from the background.

CALL	Src	Call system routine Src	Int

Fault using the vector at 128 + (Src mod 64). Src must be an integer unless the U flag is set.

4.9 NAME CACHE OPERATIONS

ENTER	Src, Rs	Enter(Src) Into Rs	All but CFut
XLATE	Rs, Dst, C	Dst ← lookup in Rs	All but CFut
PROBE	Src, Rs	Rd ← Bool:Src is in Rs	All but CFut

The Enter instruction enter Rs and Src into the associative table so that associative_lookup(Src)=Rs. That is, Src is the key and Rs is the data. The slot used is picked at random except when associative_lookup(Src) already existed, in which case the old value is overwritten.

Xlate recovers the value entered using the key Rs. An XLATE fault is signalled if no entry was found in the table or if the associated data value for Rs is NIL. The constant field C provides a way for the XLATE exception code to know what circumstances surrounded the failed translation so it can behave appropriately. When XLATE'ing into an address register the key being XLATE'd is written into the corresponding ID register. The probe instruction is similar to xlate except it does not fault if Rs in not found. If Rs is there, Dst ←Lookup(Rs), else Dst ← NIL.

INVAL	Invalidate address register

Invalidate all relocatable address registers (ones with the R bit set) on both priority levels by copying the R bit into the I bit.

4.10 MDP BUGS

The current implementation of the Message Driven Processor has a small number of bugs that the low-level programmer must be conscious of. The file "~mdp/doc/bugs_and_workarounds" indicates the known bugs and how to work around them.

Chapter 5

PROGRAMMING EXAMPLES

In this section we provide several simple example programs written using the assembler that is incorporated into MDPSim. We also discuss how to run the assembler and the use of standard include files.

5.1 THE FORM OF A TYPICAL PROGRAM

A typical MDP assembly language program is of the following form:

```
        INCLUDE "/home/jm/mdp/include/hw.mdp"

;;; Define the location and size of the message queues
LABEL qbm0_base = $0200
LABEL qbm0_size = $0100
LABEL qbm1_base = $0300
LABEL qbm1_size = $0080

;;; Define the location and size of the name cache
LABEL tbm_base  = $0380
LABEL tbm_size  = $0080

<more labels>

        MODULE
        ORG reset_background_ip

        <initialize external memory>
        <initialize name cache>
        <initialize the network interface>

        <application code>

        <message handlers>

        <fault handlers>

        <call handlers>

        ORG fault_vec_addr_p0
        <priority zero fault vector>

        ORG fault_vec_addr_p1
        <priority one  fault vector>

        ORG syscall_vec_addr
        <system call table>

        END
```

Quite frequently the programmer will use a set of standard fault handlers and associated vectors. The file "~mdp/include/std_flt.i" contains a handler that saves the state of the processor core into memory and then idles, and a set of simple stubs that record which fault occurs and then calls this main routine. This state can be inspected by utilities in the monitor. If the user wishes to supply their own handler for certain faults, then they can do so explicitly during program initialization. In this case the program has the following form:

```
        INCLUDE "/home/jm/mdp/include/hw.i"

;;; Define the location and size of the message queues
LABEL qbm0_base = $0200
LABEL qbm0_size = $0100
LABEL qbm1_base = $0300
LABEL qbm1_size = $0080

;;; Define the location and size of the name cache
LABEL tbm_base  = $0380
LABEL tbm_size  = $0080

<more labels>

        MODULE
        ORG reset_background_ip

        <initialize external memory>
        <override certain trap handlers>
        <initialize name cache>
        <initialize the network interface>

        <application code>

        <message handlers>

        <additional fault handlers>

        <additional call handlers>

        INCLUDE "/home/jm/mdp/include/std_flt.i"

        END
```

This program can be loaded directly into the MDP instruction level simulator MDPSim. Alternatively, it can be assembled using MDPSim and then loaded into the register transfer level simulator or onto the J-Machine. Assembling a program for the J-Machine is accomplished by

```
unix> MDPSim -o test.bin test.mdp
```

and then this program can be loaded and run using jmon. The use of MDPSim is fully documented in CVA Memo 38 "Message-Driven Processor Simulator". All of the examples provided below are included in the directory "~mdp/examples" so that you can run them for yourself and trace their execution.

5.2 INITIALIZING THE MDP

This program demonstrates the steps required to initialise an MDP. It concentrates on these key steps and little else. We include the mdp assembly language program itself and a jmon script for running this program. Please refer to the jmon manual for more information on jmon scripts. It assumes that the front end has placed the node-id for every node in a known location in memory. The program will be downloaded through the diagnostic port to all the nodes at one time and then this location will be written on each node in turn.

This program accesses registers for both P0 and P1 from the background. A special syntax is used to specify which priority is to be used. In this code, the embedded comments clarify which priority is being used. Accessing registers among the priorities is clarified in a later example.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
; Michael Noakes                 init-mdp.mdp                          Oct  1, 1991 ;
;                                                                                  ;
; This program demonstrates the key steps that must occur to initialize an         ;
; MDP.  It is assumed that the front-end supplies the NNR in memory.               ;
;                                                                                  ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

INCLUDE "/home/jm/mdp/include/hw.i"

;;; Define the location and size of the message queues
LABEL qbm0_base = $0200
LABEL qbm0_size = $0100
LABEL qbm1_base = $0300
LABEL qbm1_size = $0080

;;; Define the location and size of the name cache
LABEL tbm_base  = $0380
LABEL tbm_size  = $0080

;;;  Do not change these locations without changing JMON scripts also
LABEL NnrTable  = $0120
LABEL NodeNNR   =     0    ; This node's NNR or NIL
LABEL MinNNR    =     1    ; The min NNR in the cube
LABEL MaxNNR    =     2    ; The min NNR in the cube

;----------------------------------------------------------

        MODULE
        ORG reset_background_ip

        ; initialize the refresh timer and error counter
        dc      emi_rtc                  ; address of timer register
        move    r0, r1
        dc      emi_rtc_init             ; initial count
        move    r0, [r1, a0]             ; store into timer

        dc      emi_erc                  ; The error count register
        move    0, r1
        move    r1, [r0, a0]

        ;;; Initialize the QBM and QHL registers for P0
        dc      ADDR:(qbm0_base << addr_base_pos) |             0
        move    r0, qhl
        dc      ADDR:(qbm0_base << addr_base_pos) | (qbm0_size - 1)
        move    r0, qbm

        ;;; Initialize the QBM and QHL registers for P1
        dc      ADDR:(qbm1_base << addr_base_pos) |             0
        move    r0, qhl'
        dc      ADDR:(qbm1_base << addr_base_pos) | (qbm1_size - 1)
        move    r0, qbm'

        ;;; Initialize the name cache
        dc      ADDR:(tbm_base  << tbm_base_pos)  | (tbm_size - 1)
        move    r0, tbm

        ;;; Set the SEND and EARLY fault handler specially
        dc      ADDR:(fault_vec_addr_p0 << addr_base_pos) | fault_vec_len
```

```
        move    r0, a1
        dc      ADDR:(fault_vec_addr_p1 << addr_base_pos) | fault_vec_len
        move    r0, a2

        dc      IP:ip_u | ip_f | (RETRY  << ip_offset_pos) | ip_a0_absolute
        move    r0, [fault_send,  a1]
        move    r0, [fault_send,  a2]

        move    r0, [fault_early, a1]
        move    r0, [fault_early, a2]

        ;;; Fetch the node-id and store it in the NNR
        dc      NodeNNR
        move    [r0, a0], r1
        move    r1, NNR

        ;;; Send messages to self to initialize the routers
        dc      MSG:(INITR << addr_base_pos) | 1
        send2e  r1, r0, 1
        send2e  r1, r0, 0

        ;;; Accept interrupts
        move    0, r0
        move    r0, I

        ;;; Wait for messages to run through the datapath
        move    4, r0
Idle1:  sub     r0, 1, r0
        bnz     r0, ^Idle1

Done:   br      ^Done

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;                                                                   ;
        ; Message INITR()                                                   ;
        ;                                                                   ;
        ; Sent by a node to initialize its router datapath.                 ;
        ;                                                                   ;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

INITR:  suspend

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;                                                                   ;
        ; The custom FAULT handlers                                         ;
        ;                                                                   ;
        ; RETRY:  Waste some time while restarting the instruction          ;
        ;                                                                   ;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;;; Subtract a phase from FIP the sneaky way and retry
RETRY:  move    r0, fop0
        move    fip, r0
        rot     r0, -ip_p_pos, r0
        sub     r0, 1, r0
        rot     r0,  ip_p_pos, r0
        move    r0, fip
        move    fop0, r0
        ldipr   fip
```

```
; Include standard fault vectors
INCLUDE "/home/jm/mdp/include/std_flt.i"

        END
```

Here is a jmon script that executes this program though there is no interesting data generated by running this one. The script is included to clarify how to initialise this program.

```
/*****************************************************************/
/*                                                               */
/* Mike Noakes         init-mdp.j                  Oct 1, 1991 */
/*                                                               */
/* This script initializes and runs a program that demonstrates */
/* the basic process required to initialize the MDP.             */
/*                                                               */
/*****************************************************************/

/* Include the standard jmon utilities */
include "std_util.j"

/* The address where the MDP expects to find its NNR     */
NodeNNR = $120;

/* Run the program on a machine of the indicated size    */
defun InitMdp (xsize ysize zsize) {

 make_mdp xsize ysize zsize;          /* Make the machine */

 select_all;
 reset;
 load "init-mdp.bin";                 /* Load the program */

 /* Select each node in turn and store its NNR */
 for (i = 0; i < _NNODES; i = i + 1) {
   select i;
   sm NodeNNR (nodeid_to_nnr i);
 }

 select_all;

 /* Start processor and run cycle until first instruction */
 run   1;

 /* Watch the register file at the current priority */
 if (_JMI == "RTL ")
    .watch cur_state ;

 run  30;
 halt;
}

InitMdp 1 1 1;
quit;
```

5.3 INITIALIZING THE NNRs

This program example demonstrates the general principles of initialising an MDP and one method for assigning node-ids to every node. This program runs on a J-Machine without a network interface to the host machine; it relies on the use of several memory locations known to both the program and the jmon interface to communicate the configuration of the machine.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
; Michael Noakes                 init-nnr.mdp                     Oct  4, 1991 ;
;                                                                              ;
; This program demonstrates the initialization of an MDP and implements a      ;
; simple distributed scheme for assigning node-ids to each node.  This         ;
; program is loaded onto every node.  One processor is selected to be the      ;
; driver node.  Every node learns the min and max node-ids of the machine      ;
; using memory locations known to both the MDP and the jmon front-end.         ;
; A third location indicates whether this node is to be the master.            ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


        INCLUDE "/home/jm/mdp/include/hw.i"

;;; Define the location and size of the message queues
LABEL qbm0_base       = $0200
LABEL qbm0_size       = $0100
LABEL qbm1_base       = $0300
LABEL qbm1_size       = $0080

;;; Define the location and size of the name cache
LABEL tbm_base        = $0380
LABEL tbm_size        = $0080

;;;  Do not change these locations without changing JMON scripts also
LABEL NnrTable        = $0120
LABEL NnrTableLength  =     4
LABEL MinNNR          =     0    ; The min NNR in the cube
LABEL NodeNNR         =     1    ; This node's NNR or NIL
LABEL MaxNNR          =     2    ; The max NNR in the cube
LABEL InitMsgs        =     3    ; The number of init messages

;------------------------------------------------------------

        MODULE
        ORG reset_background_ip

        ; initialize the refresh timer and error counter
        dc      emi_rtc                 ; address of timer register
        move    r0, r1
        dc      emi_rtc_init            ; initial count
        move    r0, [r1, a0]            ; store into timer

        dc      emi_erc                 ; The error count register
        move    0, r1
        move    r1, [r0, a0]

        ;;; Initialize the QBM and QHL registers
        dc      ADDR:(qbm0_base << addr_base_pos) |              0
        move    r0, qhl
        dc      ADDR:(qbm0_base << addr_base_pos) | (qbm0_size - 1)
        move    r0, qbm

        dc      ADDR:(qbm1_base << addr_base_pos) |              0
        move    r0, qhl'
        dc      ADDR:(qbm1_base << addr_base_pos) | (qbm1_size - 1)
        move    r0, qbm'
```

```
        ;;; Initialize the name cache
        dc      ADDR:(tbm_base  << tbm_base_pos)  | (tbm_size - 1)
        move    r0, tbm

        ;;; Set the SEND and EARLY fault handler specially
        dc      ADDR:(fault_vec_addr_p0 << addr_base_pos) | fault_vec_len
        move    r0, a1
        dc      ADDR:(fault_vec_addr_p1 << addr_base_pos) | fault_vec_len
        move    r0, a2

        dc      IP:ip_u | ip_f | (RETRY << ip_offset_pos) | ip_a0_absolute
        move    r0, [fault_send,  a1]
        move    r0, [fault_send,  a2]

        move    r0, [fault_early, a1]
        move    r0, [fault_early, a2]

        ;;; Accept interrupts
        move    0, r0
        move    r0, I

        ;;; Wait until the NNR is set.  This can be done by front-end or message
        dc      addr:(NnrTable << addr_base_pos) | NnrTableLength
        move    r0, a1
Wait:   move    [NodeNNR, a1], r1
        bnil    r1, ^Wait

        ;;; Store it in the NNR
        move    r1, NNR

        ;;; Send messages to self to initialize the routers
        dc      MSG:(INITR << addr_base_pos) | 1
        send2e  r1, r0, 1
        send2e  r1, r0, 0

        ;;; Fetch the Max NNR
        move    [MaxNNR, a1], r2

        ;;;;;;;;;;;;;;;;;;  Check the Z-dimension ;;;;;;;;;;;;;;;;;;;;

        ;;; IF this node "above" the origin node?
ChkZ:   sub     r1, [MinNNR, a1], r3
        and     r3, $3FF, r3
        bnz     r3, ^ChkY

        ;;; AND not on the top board
        dc      int:$7C00
        sub     r2, r1, r3
        and     r3, r0, r3
        bz      r3, ^ChkY

        ;;; THEN send the SETNNR to the node above
        dc      1 << 10
        add     r1, r0, r3
        dc      msg:(SETNNR << addr_base_pos) | 4
        send2   r3, r0, 0
        send2   [MinNNR, a1], r3, 0
        sende   r2, 0

        ;;;;;;;;;;;;;;;;;;  Check the Y-dimension ;;;;;;;;;;;;;;;;;;;;
```

```
        ;;; IF this node in the first column
ChkY:   dc      $1F
        sub     r1, [MinNNR, a1], r3
        and     r3, r0, r3
        bnz     r3, ^ChkX

        ;;; AND not at the top of the column
        dc      int:$03E0
        sub     r2, r1, r3
        and     r3, r0, r3
        bz      r3, ^ChkX

        ;;; THEN send the SETNNR to the node in the positive Y-dim
        dc      1 << 5
        add     r1, r0, r3
        dc      msg:(SETNNR << addr_base_pos) | 4
        send2   r3, r0, 0
        send2   [MinNNR, a1], r3, 0
        sende   r2, 0

        ;;;;;;;;;;;;;;;;;;  Check the X-dimension ;;;;;;;;;;;;;;;;;;;;;

        ;;; IF this node is not in the last column
ChkX:   dc      int:$001F
        sub     r2, r1, r3
        and     r3, r0, r3
        bz      r3, ^Done

        ;;; THEN send the SETNNR to the node in the positive X-dim
        add     r1, 1, r3
        dc      msg:(SETNNR << addr_base_pos) | 4
        send2   r3, r0, 0
        send2   [MinNNR, a1], r3, 0
        sende   r2, 0

Done:   br      ^Done

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;                                                                 ;
        ; Message INITR()                                                 ;
        ;                                                                 ;
        ; Sent by a node to initialize its router datapath.               ;
        ;                                                                 ;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

INITR:  suspend

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;                                                                 ;
        ; Message SETNNR(minnnr nodennr maxnnr)                           ;
        ;                                                                 ;
        ; Sent by a neighbor to pass us our NNR. Copy the args into memory ;
        ;                                                                 ;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

SETNNR: dc      addr:(NnrTable << addr_base_pos) | NnrTableLength
        move    r0, a1

        ;;; Copy the arguments from the message into memory
        move    [1, a3], r0
```

```
        move    r0, [MinNNR,  a1]
        move    [2, a3], r0
        move    r0, [NodeNNR, a1]
        move    [3, a3], r0
        move    r0, [MaxNNR,  a1]

        ;;; Increment the counter (Zeroed by the front end)
        move    [InitMsgs, a1], r0
        add     r0, 1, r0
        move    r0, [InitMsgs, a1]

        suspend

        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;                                                                   ;
        ; The custom FAULT handlers                                         ;
        ;                                                                   ;
        ; RETRY:  Waste some time while restarting the instruction.  This   ;
        ; handler must disable interrupts while it computes the previous    ;
        ; IP because it needs to use fop0 to store R0 and yet can be run    ;
        ; in the background as well as at P0.  Since background and P0      ;
        ; share FOP0 there is a risk of it getting clobbered.               ;
        ;                                                                   ;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        ;;; Turn off interrupts without using R0
RETRY:  bt      r0, ^RETRY1
        xor     r0, 1, r0
        move    r0, I
        xor     r0, 1, r0
        br      ^RETRY2
RETRY1: move    r0, I

        ;;; Subtract a phase from FIP the sneaky way using R0
RETRY2: move    r0, fop0
        move    fip, r0
        rot     r0, -ip_p_pos, r0
        sub     r0, 1, r0
        rot     r0,  ip_p_pos, r0
        move    r0, fip
        move    fop0, r0

        ;;; Turn interrupts back on without clobbering R0 and return
        bt      r0, ^RETRY3
        move    r0, I
        ldipr   fip

RETRY3: xor     r0, 1, r0
        move    r0, I
        xor     r0, 1, r0
        ldipr   fip

; Include standard fault vectors
INCLUDE "/home/jm/mdp/include/std_flt.i"

        END
```

An interesting point to note about this example is that the retry handler, used to delay in the event of a send fault, is considerably more complicated than in the previous example. This handler is more robust than the previous and has been included to demonstrate a difficult aspect of taking faults in the background. Most trap handlers need to free one or two of the general registers for scratch use and so there must be

copied somewhere safe. One popular place is the fop0 and fop1 registers if they are available and another is the low slots in the priority switchable memory since they can be accessed directly. Unfortunately, the background and priority zero share both of these areas assuming that the P-bit is still set to 0. This is a potential risk when working in the background because it is generally possible that a dispatch to priority zero might occur and then that message might also take a fault.

One could take the stance that the background should be used very carefully and should not generate faults. This is perhaps a good goal, and certainly this example could be written that way, but it is very tempting to use background and it is often quite natural to do interesting things there. This retry handler demonstrates a very neat trick for disabling interrupts before saving R0 without affecting R0. It relies on the observation that the sense of a boolean flag, i.e. the I-bit, depends only on the low bit of the value written to it. If R0 already contains a 1 in the least-significant bit then we need just write R0 to I. If it contains 0 in the lsb, then we can reversibly flip it to 1 using the XOR instruction. A similar process enables interrupts at the end of the handler. Of course, this handler must run in unchecked mode.

Here is a jmon script to run this application on the J-Machine hardware. It prepares the global flags and verifies that the correct values NNRs were installed.

```
/*****************************************************************/
/*                                                               */
/* Mike Noakes          init-nnr.j                  Oct 4, 1991 */
/*                                                               */
/* This script initializes and runs a program that demonstrates */
/* a simple technique to distribute NNRs.                        */
/*                                                               */
/*****************************************************************/

/* Include the standard jmon utilities */
include "std_util.j"

/* The address where the MDP expects to find its NNR     */
MinNNR   = $120;
NodeNNR  = $121;
MaxNNR   = $122;
InitMsgs = $123;

/* Run the program on a machine of the indicated size    */
defun InitNNR (xsize ysize zsize) {

 make_mdp xsize ysize zsize;          /* Make the machine */

 select_all;
 reset;
 load "init-nnr.bin";                 /* Load the program */

 sm MinNNR   0:0;                     /* Set to NIL       */
 sm NodeNNR  0:0;                     /* Set to NIL       */
 sm MaxNNR   0:0;                     /* Set to NIL       */
 sm InitMsgs 1:0;                     /* Set to zero      */

 /* Set node 0 specially */
 select 0;
 sm MinNNR   (nodeid_to_nnr 0);
 sm NodeNNR  (nodeid_to_nnr 0);
 sm MaxNNR   (nodeid_to_nnr _NNODES - 1);
 sm InitMsgs 1:1;

 /* Run */
 select_all;
 run   10000;
 halt;

 /* Check the answers */
```

```
for (i = 0; i < _NNODES; i = i + 1) {
  select i;

  if ((nodeid_to_nnr i) != (rmem NodeNNR))
     printf "Node %2d has the wrong NNR: <%d %d %d> rather than <%d %d %d>\n"
            i

            (NNRx (rmem NodeNNR))
            (NNRy (rmem NodeNNR))
            (NNRz (rmem NodeNNR))

            (NNRx (nodeid_to_nnr i))
            (NNRy (nodeid_to_nnr i))
            (NNRz (nodeid_to_nnr i));

  if (:?(rmem InitMsgs) != 1)
     printf "Node %2d received %d init messages, not 1 as expected\n"
            i
            :?(rmem InitMsgs);
 }
}
```

5.4 ACCESSING OTHER PRIORITIES

One issue that often causes people to stumble is the appropriate syntax for accessing registers in each priority from each priority. The following code demonstrates the use of the ‘b’ and ‘'’ syntax for describing accesses to registers at other priorities as well as the use of the priority-switchable memory.

In this program the registers R3, QHL, and relative addresses 0 and $40 are accessed from each priority using all possible combinations of the ‘b’ and ‘'’ suffixes. Specifying a suffix of ‘b’ sets the background flag and specifying ‘'’ sets the priority flag. If the selected register is implemented at all three priorities, as R3 is, then the background flag is XORed with the processor's B-bit. If the result is 1 then the background register is selected and the priority flag is ignored. If the result is 0 then the priority flag is used. If it is 0 then the register should be taken from the priority indicated by the processor's P-bit. A priority flag of 1 selects the opposite priority. If the register does not occur in the background, e.g. QHL, then the background flag is ignored and the priority flag selects the priority relative to the P-bit.

This program does two things that are unusual and should normally be avoided. Most seriously, it executes a small number of instructions in the background with P set to 1. We define the background to be B = 1 and P = 0 for the reason demonstrated by this code; the P bit will affect the register selected in some cases and code would be hard to understand if programmers ignore this convention. It also switches priority by explicitly modifying the B and P flags rather than by sending messages. This was done to help concentrate on the effects of altering the various flags. It is not recommended practice for most programs but the effects are interesting and it is conceivable that a user might have a purpose for doing this. Note carefully that it is necessary to prepare the instruction pointer before changing the flags.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
; Michael Noakes                    pri.mdp                      Oct  1, 1991 ;
;                                                                              ;
; This program demonstrates the use of register mode addressing to select     ;
; registers in other priorities.  It also demonstrates the ability to switch  ;
; priorities without sending messages by manipulating the B and P bit         ;
; explicitly.                                                                  ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        INCLUDE "/home/jm/mdp/include/hw.i"

        MODULE
        ORG reset_background_ip
```

```
        ;;; Place known values into each of the registers
        move    1, r0
        move    r0, r3
        move    2, r0
        move    r0, r3b
        move    3, r0
        move    r0, r3b'

        dc      addr:($400 << 10)
        move    r0, qhl
        dc      addr:($500 << 10)
        move    r0, qhl'

        move    0, r0
        move    r0, [r0, a0]
        dc      $40
        move    r0, [r0, a0]

        ;;; From Background   B = 1   P = 0   A0-Absolute
test0:  readr   r3,    r0               ; Fetch R3BK  Use B, ignore P
        readr   r3b,   r0               ; Fetch R3P0  Flip B, current P
        readr   r3b',  r0               ; Fetch R3P1  Flip B, other P
        readr   r3',   r0               ; Fetch R3BK  Use B, ignore P

        readr   qhl,   r0               ; Fetch QHLP0.  Current P
        readr   qhlb,  r0               ; Fetch QHLP0.  Current P
        readr   qhl',  r0               ; Fetch QHLP1.  Other P
        readr   qhlb', r0               ; Fetch QHLP1.  Other P

        move    0, r0
        move    [r0, a0], r1            ; Fetch physical address 0
        dc      $40
        move    [r0, a0], r1            ; Fetch physical address $40

        ;;; Switch to the background with P = 1.
        ;;; This is not recommended practice
        move    1, r0
        move    r0, P

test1:  readr   r3,    r0               ; Fetch R3BK  Use B, ignore P
        readr   r3b,   r0               ; Fetch R3P1  Flip B, current P
        readr   r3b',  r0               ; Fetch R3P0  Flip B, other P
        readr   r3',   r0               ; Fetch R3BK  Use B, ignore P

        readr   qhl,   r0               ; Fetch QHLP1.  Current P
        readr   qhlb,  r0               ; Fetch QHLP1.  Current P
        readr   qhl',  r0               ; Fetch QHLP0.  Other P
        readr   qhlb', r0               ; Fetch QHLP0.  Other P

        move    0, r0
        move    [r0, a0], r1            ; Fetch physical address $40
        dc      $40
        move    [r0, a0], r1            ; Fetch physical address $0

        ;;; Switch to Priority 0 explicitly.  B = 0, P = 0.  A0-Absolute
        dc      IP:ip_u | (test2 << ip_offset_pos) | ip_a0_absolute
        move    r0, ipb'
        move    0, r0
        move    r0, P
        move    r0, B
```

```
        ;;; This code runs at priority zero
test2:  readr   r3,     r0              ; Fetch R3P0  Current P
        readr   r3b,    r0              ; Fetch R3BK  Flip B, ignore P
        readr   r3b',   r0              ; Fetch R3BK  Flip B, ignore P
        readr   r3',    r0              ; Fetch R3P1  Other P

        readr   qhl,    r0              ; Fetch QHLP0.  Current P
        readr   qhlb,   r0              ; Fetch QHLP0.  Current P
        readr   qhl',   r0              ; Fetch QHLP1.  Other P
        readr   qhlb',  r0              ; Fetch QHLP1.  Other P

        move    0, r0
        move    [r0, a0], r1            ; Fetch physical address $0
        dc      $40
        move    [r0, a0], r1            ; Fetch physical address $40

        ;;; Switch to Priority 1 explicitly.  B = 0, P = 1.  A0-Absolute
        dc      IP:ip_u | (test3 << ip_offset_pos) | ip_a0_absolute
        move    r0, ip'
        move    1, r0
        move    r0, P

        ;;; This code runs at priority one
test3:  readr   r3,     r0              ; Fetch R3P1  Current P
        readr   r3b,    r0              ; Fetch R3BK  Flip B, ignore P
        readr   r3b',   r0              ; Fetch R3BK  Flip B, ignore P
        readr   r3',    r0              ; Fetch R3P0  Other P

        readr   qhl,    r0              ; Fetch QHLP1.  Current P
        readr   qhlb,   r0              ; Fetch QHLP1.  Current P
        readr   qhl',   r0              ; Fetch QHLP0.  Other P
        readr   qhlb',  r0              ; Fetch QHLP0.  Other P

        move    0, r0
        move    [r0, a0], r1            ; Fetch physical address $40
        dc      $40
        move    [r0, a0], r1            ; Fetch physical address $0

        ;;; All access done
idle:   br      ^idle

END
```

This program was assembled by typing

```
unix> MDPSim -o pri.bin pri.mdp
```

at the command line while in the examples directory. There is also a jmon script file that runs this example while "watching" the register file under the register-transfer-level simulator. The RTL simulator is too slow to be valuable for general program development but it is quite adequate for this program and it does do a reasonable job of describing the register file. Here is the jmon script that can be used to run this experiment:

```
/*****************************************************************/
/*                                                               */
/* Mike Noakes            pri.j                    Oct 1, 1991 */
/*                                                               */
/* A simple jmon script to run a program that demonstrates the   */
/* MDPSim assembler syntax for accessing priorities.  This       */
/* script relies on special commands in the RTL simulator to     */
/* view the register file.                                       */
```

```
/*                                                                    */
/**********************************************************************/

/* Load the program */
load "pri.bin";

/* Use low-level commands built into the RTL simulator          */
if (_JMI == "RTL ") {
   .reset                    /* Reset the MDP and issue "GO"   */
   .dec                      /* Input base is decimal          */
   .watch all_state          /* Watch the entire register file */
   .execute until 205        /* Execute until cycle 205        */
}

/* Exit JMON */
quit;
```

It can be executed by doing

```
unix> rtl -r pri.j > pri.log
```

which explicitly requests the RTL simulator when running jmon and which will dump the output to the file pri.log. You will want to widen your emacs a little compared to the typical default to view this file.

5.5 LONG JUMPS

It was noted that the range available for immediate branches of the form

```
        bt      r0, ~Target
```

is -64 to +63 words. This span is sufficient for typical application code especially in an environment in which we expect the total method length to be on this order, but is occasionally inadequate. There are two standard methods for making jumps to target to targets further away than this; compute the desired value for the instruction pointer and load it directly, or compute the relative distance into a register and branch with this amount.

```
        ;;; Compute the new IP.  Run in checked mode and a0_absolute
        dc      ip:(Target << ip_offset_pos) | ip_a0_absolute
        ldip    r0

        ;;; Compute the branch distance
        dc      Target - (* + 2)
        br      r0

        <code>

Target: <more code>
```

There is little to chose between these approaches. The principle disadvantage to the former technique is that the user is then responsible for preserving the various flags that are stored in the IP. Of course, this is a good solution if you wish to explicitly alter these flags before running the next block. The second solution avoids this issue but is, perhaps, a little quirkier. The '*' is a pseudo-variable maintained by the MDPSim assembler that contains the value of the current code offset. The factor of 2 is added to compensate for the instruction prefetch mechanism of the MDP.

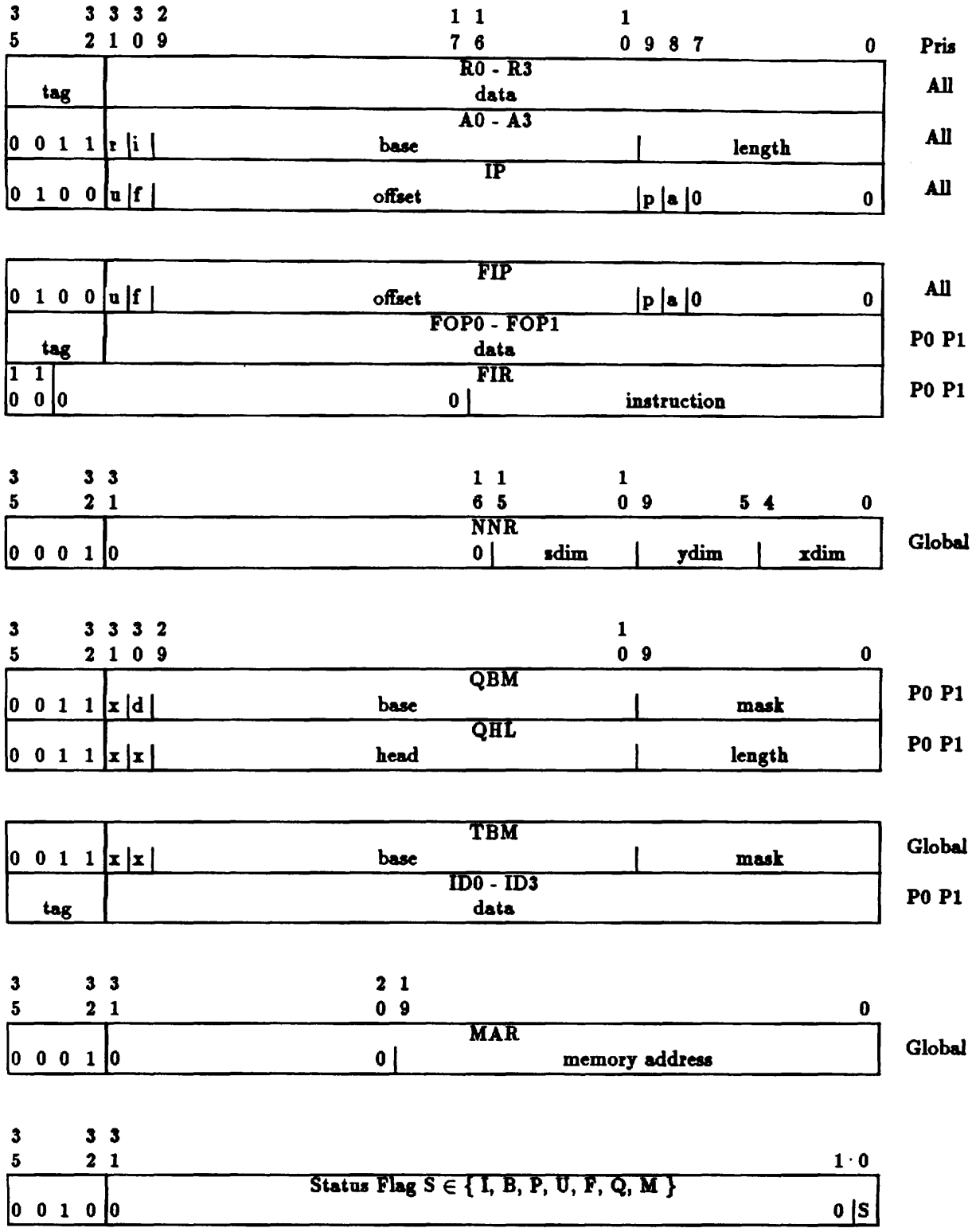
Chapter 6

Appendix A

This appendix collects the key tables in one place. It shows the format of the 13 defined types, the format of the register file, summarises the instruction set of the processor, and the set and priority of the 19 defined faults.

3 3	3 3 2	1 1	1		
5	2 1 0 9	7 6	0 9 8 7	1 0	
0 0 0 0	Value(0 = NIL)				SYM
0 0 0 1	Two's Complement Value				INT
0 0 0 1	0			0 b	BOOL
0 0 1 1	r i	base	length		ADDR
0 1 0 0	u f	offset	p a 0	0	IP
0 1 0 1	u f	offset	length		MSG
0 1 1 0	user-defined				CFUT
0 1 1 1	user-defined				FUT
1 0 0 0	user-defined				TAG8
1 0 0 1	user-defined				TAG9
1 0 1 0	user-defined				TAGA
1 0 1 1	user-defined				TAGB
1 1 0 0	first instruction		second instruction		INST0
1 1 0 1	first instruction		second instruction		INST1
1 1 1 0	first instruction		second instruction		INST2
1 1 1 1	first instruction		second instruction		INST3

Register Set



I:	Interrupt Mask	0: Interrupts Allowed	1: Interrupts Disabled
B:	Background Execution	0: Message	1: Background
P:	Priority Level	0: Level 0	1: Level 1
U:	Unchecked Mode	0: Checked	1: Unchecked
F:	Faults Disabled	0: Normal	1: Faults Disallowed
Q:	Queue Wrap Around	0: A3 Points Into Memory	1: A3 Points Into Queue
M:	Message Flag	0: Message Send Complete	1: Message Being Sent

General Movement and Type Instructions			Source Types
READ	Src, Rd	Rd ← Src.	All but CFut
READR	Src, Rd	Rd ← Src.	All but CFut
WRITE	Rs, Dst	Dst ← Rs.	All
WRITER	Rs, Dst	Dst ← Rs.	All
LDIP	Src	IP ← Src.	IP
LDIPR	Src	IP ← Src.	IP
CHECK	Rs, Src, Rd	Rd ← BOOL:tag(Rs) == Src	All,Int
RTAG	Src, Rd	Rd ← INT:tag(Src)	All but CFut
WTAG	Rs, Src, Rd	Rd ← Src:Rs	All,Int
Arithmetic and Logical Instructions			
ADD	Rs, Src, Rd	Rd ← Rs + Src	Int,Int
CARRY	Rs, Src, Rd	Rd ← Carry(Rs + Src)	Int,Int
SUB	Rs, Src, Rd	Rd ← Rs - Src	Int,Int
NEG	Rs, Src	Rd ← -Src	Int
MUL	Rs, Src, Rd	Rd ← Low 32 bits of Rs*Src	Int,Int
MULH	Rs, Src, Rd	Rd ← High 32 bits of Rs*Src	Int,Int
ASH	Rs, Src, Rd	Rd ← Rs << Src (arithmetic)	Int,Int
LSH	Rs, Src, Rd	Rd ← Rs << Src (logical)	Int,Int
ROT	Rs, Src, Rd	Rd ← Rs rotate left Src	Int,Int
FFB	Src, Rd	Rd ← Find First Bit	Int
AND	Rs, Src, Rd	Rd ← Rs AND Src	Int,Int Bool,Bool
NOT	Src, Rd	Rd ← NOT Src	Int Bool
OR	Rs, Src, Rd	Rd ← Rs OR Src	Int,Int Bool,Bool
XOR	Rs, Src, Rd	Rd ← Rs XOR Src	Int,Int Bool,Bool
GE	Rs, Src, Rd	Rd ← Bool:Rs $\geq$ Src	Int,Int Bool,Bool
GT	Rs, Src, Rd	Rd ← Bool:Rs > Src	Int,Int Bool,Bool
LE	Rs, Src, Rd	Rd ← Bool:Rs $\leq$ Src	Int,Int Bool,Bool
LT	Rs, Src, Rd	Rd ← Bool:Rs < Src	Int,Int Bool,Bool
EQUAL	Rs, Src, Rd	Rd ← Bool:Rs = Src (Data)	Int,Int Bool,Bool Sym,Sym
NEQUAL	Rs, Src, Rd	Rd ← Bool:Rs $\neq$ Src (Data)	Int,Int Bool,Bool Sym,Sym
EQ	Rs, Src, Rd	Rd ← Bool:Rs = Src (Pointer)	All but CFut or Fut
NEQ	Rs, Src, Rd	Rd ← Bool:Rs $\neq$ Src (Pointer)	All but CFut or Fut
Branches			
BR	Src	Branch forward Src words	Int
BZ	Rs, Src	Branch if Data(Rs) = 0	Int,Int
BNZ	Rs, Src	Branch if Data(Rs) $\neq$ 0	Int,Int
BF	Rs, Src	Branch if Bit0(Rs) = 0	Bool,Int
BT	Rs, Src	Branch if Bit0(Rs) = 1	Bool,Int
BNIL	Rs, Src	Branch if Rs = NIL	All but CFut,Int
BNNIL	Rs, Src	Branch if Rs $\neq$ NIL	All but CFut,Int
Network Instructions			
SEND	Src, P	Send Src at priority P	All but CFut
SENDE	Src, P	Send Src and terminate	All but CFut
SEND2	Src, Rs, P	Send Src then Rs	All but CFut
SEND2E	Src, Rs, P	Send Src then Rs and terminate	All but CFut
Special Instructions			
NOP		No Operation	
SUSPEND		Terminate Thread	
CALL	Src	Call system routine Src	Int
Associative Lookup Table Instructions			
ENTER	Src, Rs	Enter(Src) Into Rs	All but CFut
XLATE	Rs, Dst, C	Dst ← lookup in Rs	All but CFut
PROBE	Src, Rs	Rd ← Bool:Src is in Rs	All but CFut
INVAL		Invalidate address register	

Name	Args	Encoding				Faults
NOP		000000	00	00	0000000	
READ	Src, Rd	000001	Rd	i	Src	Cfut
WRITE	Rs, Dst	000010	i	Rs	Dst	
READR	Src, Rd	000011	Rd	00	Src	Cfut
WRITER	Rs, Dst	000100	00	Rs	Dst	Type, Cfut, Fut, Tag*
RTAG	Src, Rd	000101	Rd	i	Src	Cfut
WTAG	Rs, Src, Rd	000110	Rd	Rs	Src	Type, Cfut, Fut, Tag*
LDIP	Src	000111	i	00	Src	Type, Cfut, Fut, Tag*
LDIPR	Rs, Dst	001000	00	00	Src	Type, Cfut, Fut, Tag*
CHECK	Rs, Src, Rd	001001	Rd	Rs	Src	Type, Cfut, Fut, Tag*
CARRY	Rs, Src, Rd	001010	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
ADD	Rs, Src, Rd	001011	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
SUB	Rs, Src, Rd	001100	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
MULH	Rs, Src, Rd	001110	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
MUL	Rs, Src, Rd	001111	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
ASH	Rs, Src, Rd	010000	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
LSH	Rs, Src, Rd	010001	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
ROT	Rs, Src, Rd	010010	Rd	Rs	Src	Type, Cfut, Fut, Tag*
AND	Rs, Src, Rd	011000	Rd	Rs	Src	Type, Cfut, Fut, Tag*
OR	Rs, Src, Rd	011001	Rd	Rs	Src	Type, Cfut, Fut, Tag*
XOR	Rs, Src, Rd	011010	Rd	Rs	Src	Type, Cfut, Fut, Tag*
FFB	Src, Rd	011011	Rd	i	Src	Type, Cfut, Fut, Tag*
NOT	Src, Rd	011100	Rd	Rs	Src	Type, Cfut, Fut, Tag*
NEG	Rs, Src	011101	Rd	Rs	Src	Type, Cfut, Fut, Tag*, Overflow
LT	Rs, Src, Rd	100000	Rd	Rs	Src	Type, Cfut, Fut, Tag*
LE	Rs, Src, Rd	100001	Rd	Rs	Src	Type, Cfut, Fut, Tag*
GE	Rs, Src, Rd	100010	Rd	Rs	Src	Type, Cfut, Fut, Tag*
GT	Rs, Src, Rd	100011	Rd	Rs	Src	Type, Cfut, Fut, Tag*
EQUAL	Rs, Src, Rd	100100	Rd	Rs	Src	Type, Cfut, Fut, Tag*
NEQUAL	Rs, Src, Rd	100101	Rd	Rs	Src	Type, Cfut, Fut, Tag*
EQ	Rs, Src, Rd	100110	Rd	Rs	Src	Cfut, Fut
NEQ	Rs, Src, Rd	100111	Rd	Rs	Src	Cfut, Fut
XLATE	Rs, Dst, C	101000	C	Rs	Dst	Cfut, Xlate
ENTER	Src, Rs	101001	00	Rs	Src	Cfut
INVAL		101010	00	00	0000000	
PROBE	Src, Rs	101101	00	Rs	Dst	Cfut
SUSPEND		110000	00	00	0000000	Early
CALL	Src	110001	i	00	Src	Type, Cfut, Fut, Tag*
SEND	Src, P	110100	P	i	Src	Cfut, Send
SENDE	Src, P	110101	P	i	Src	Cfut, Send
SEND2	Src, Rs, P	110110	P	Rs	Src	Cfut, Send
SEND2E	Src, Rs, P	110111	P	Rs	Src	Cfut, Send
BR	Src	111000	i	00	Src	Type, Cfut, Fut, Tag*
BNIL	Rs, Src	111010	i	Rs	Src	Type, Cfut, Fut, Tag*
BNNIL	Rs, Src	111011	i	Rs	Src	Type, Cfut, Fut, Tag*
BF	Rs, Src	111100	i	Rs	Src	Type, Cfut, Fut, Tag*
BT	Rs, Src	111101	i	Rs	Src	Type, Cfut, Fut, Tag*
BZ	Rs, Src	111110	i	Rs	Src	Type, Cfut, Fut, Tag*
BNZ	Rs, Src	111111	i	Rs	Src	Type, Cfut, Fut, Tag*

Name	Number	Description
CATASTROPHE	$0	Double fault,bad vector, or other catastrophe.
INTERRUPT	$1	Interrupt signalled by diagnostic interface.
QUEUE	$2	Message queue about to overflow.
SEND	$3	Send buffer full.
ILGINST	$4	Illegal instruction.
DRAMERR	$5	Double bit error in the external RAM.
INVADR	$6	Attempt to access data through address register with I bit set.
ADRTYPE	$7	The address index is not an integer.
LIMIT	$8	Attempt to access object data past limit.
EARLY	$9	Attempt to access data in message queue before it arrived.
MSG	$A	Bad message header.
XLATE	$B	XLATE missed.
OVERFLOW	$C	Integer arithmetic overflow.
CFUT	$D	Attempted operation on a word tagged CFUT.
FUT	$E	Attempted operation on a word tagged FUT.
TAG8	$F	Attempted operation on a word tagged TAG8.
TAG9	$10	Attempted operation on a word tagged TAG9.
TAGA	$11	Attempted operation on a word tagged TAGA.
TAGB	$12	Attempted operation on a word tagged TAGB.
TYPE	$13	Operand(s) with a bad tag type used in an instruction.
	$14-$1F	Reserved for future faults.

Note: If multiple faults occur simultaneously the fault vector chosen is the one that has the highest precedence. Each fault is assigned a precedence by its fault number; lower fault numbers correspond to higher precedence.